Numerical Analysis

References:

S.D. Conte & C. de Boor, *Elementary Numerical Analysis: An Algorithmic Approach*, Third edition, 1981. McGraw-Hill.

David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Vol. 23, No. 1, March 1991.

The approach adopted in this course does not assume a very high level of mathematics; in particular the level is not as high as that required to understand Conte & de Boor's book.

1. Fundamental concepts

1.1 Introduction

This course is concerned with numerical methods for the solution of mathematical problems on a computer, usually using floating-point arithmetic. Floating-point arithmetic, particularly the 'IEEE Standard', is covered in some detail. The mathematical problems solved by numerical methods include differentiation and integration, solution of equations of all types, finding a minimum value of a function, fitting curves or surfaces to data, etc. This course will look at a large range of such problems from different viewpoints, but rarely in great depth.

'Numerical analysis' is a rigorous mathematical discipline in which such problems, and algorithms for their solution, are analysed in order to establish the *condition* of a problem or the *stability* of an algorithm and to gain insight into the design of better and more widely applicable algorithms. This course contains some elementary numerical analysis, and technical terms like *condition* and *stability* are discussed, although the mathematical content is kept to a minimum. The course also covers some aspects of the topic not normally found in numerical analysis texts, such as numerical software considerations.

In summary, the purposes of this course are:

(1) To explain floating-point arithmetic, and to describe current implementations of it.

(2) To show that design of a numerical algorithm is not necessarily straightforward, even for some simple problems.

(3) To illustrate, by examples, the basic principles of good numerical techniques.

(4) To discuss numerical software from the points of view of a user and of a software designer.

1.2 Floating-point arithmetic

1.2.1 Overview

Floating-point is a method for representing real numbers on a computer.

Floating-point arithmetic is a very important subject and a rudimentary understanding of it is a pre-requisite for any modern numerical analysis course. It is also of importance in other areas of computer science: almost every programming language has floating-point data types, and these give rise to special floating-point *exceptions* such as *overflow*. So floating-point arithmetic is of interest to compiler writers and designers of operating systems, as well as to any computer user who has need of a numerical algorithm. Sections 1.2.1, 1.2.2, 1.7, 1.13 and 3.3.2 are closely based on Goldberg's paper, which may be consulted for further detail and examples.

Until 1987 there was a lot in common between different computer manufacturers' floating-point implementations, but no Standard. Since then the IEEE Standard has been adopted by some, but by no means all manufacturers and may yet grow in popularity. This Standard has taken advantage of the increased speed of modern processors to implement a floating-point model that is superior to earlier implementations because it is required to cater for many of the special cases that arise in calculations. The IEEE Standard gives an algorithm for the basic operations $(+ - * / \sqrt{})$ such that different implementations must produce the same results, i.e. in every bit. (In these notes the symbol $*$ is used to denote floating-point multiplication whenever there might be any confusion with the symbol $\times$, used in representing the value of a floating-point number.) The IEEE Standard is by no means perfect, but it is a major step forward: it is theoretically possible to prove the correctness of at least some floating-point algorithms when implemented on different machines under IEEE arithmetic. IEEE arithmetic is described in Section 1.13.

We first discuss floating-point arithmetic in general, without reference to the IEEE Standard.

1.2.2 General description of floating-point arithmetic

Floating-point arithmetic is widely implemented in hardware, but software emulation can be done although very inefficiently. (In contrast, fixed-point arithmetic is implemented only in specialised hardware, because it is more efficient for some purposes.)

In floating-point arithmetic fractional values can be represented, and numbers of greatly different magnitude, e.g. 10^{30} and 10^{-30}. Other real number representations exist, but these have not found widespread acceptance to date.

Each floating-point implementation has a *base* β which, on present day machines, is typically 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). Base 10 is often found on hand-held calculators but rarely in fully programmable computers. We define the *precision* p as the number of digits (of base β) held in a floating-point number. If d_i represents a digit then the general representation of a floating-point number is

$$\pm d_0.d_1d_2d_3\ldots d_{p-1} \times \beta^e$$

which has the value

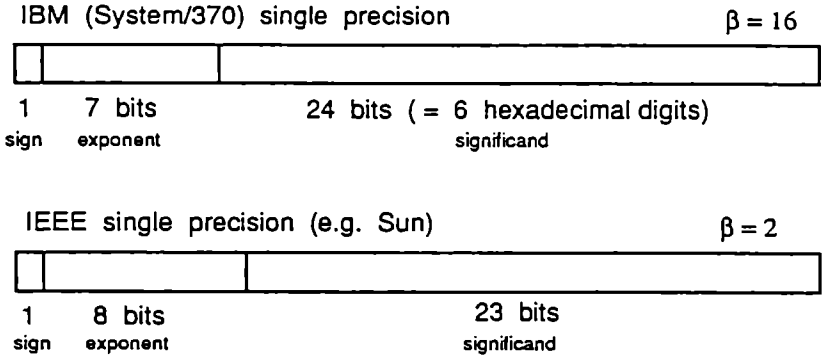
$$\pm(d_0 + d_1\beta^{-1} + d_2\beta^{-2} + \ldots + d_{p-1}\beta^{-(p-1)})\beta^e$$

where $0 \leq d_i < \beta$.

If $\beta = 2$ and $p = 24$ then 0.1 cannot be represented exactly but is stored as the approximation $1.10011001100110011001101 \times 2^{-4}$. In this case the number 1.10011001100110011001101 is the *significand* (or *mantissa*), and -4 is the *exponent*. The storage of a floating-point number varies between machine ranges. For example, a particular computer may store this number using 24 bits for the significand, 1 bit for the sign (of the significand), and 7 bits for the exponent in order to store each floating-point number in 4 bytes. Two different machines may use this format but store the significand and exponent in the opposite order; calculations might even produce the same answers but the internal bit-patterns in each word will be different.

Note also that the exponent needs to take both positive and negative values, and there are various conventions for storing these. However, all implementations have a maximum and minimum value for the signed exponent,

usually denoted by e_{max} and e_{min}.



Both of the above examples have the most significant digit of the significand on the left. Some floating-point implementations have the most significant digit on the right, while others have sign, exponent and significand in a different order.

Although not a very realistic implementation, it is convenient to use a model such as $\beta = 10$, $p = 3$ to illustrate examples that apply more generally. In this format, note that the number 0.1 can be represented as 1.00×10^{-1} or 0.10×10^{0} or 0.01×10^{1}. If the leading digit d_0 is non-zero then the number is said to be *normalized*; if the leading digit is zero then it is *denormal* (or *subnormal*). The normalized representation of any number, e.g. 1.00×10^{-1}, is unique.

In a particular floating-point implementation it is possible to use normalized numbers only, except that zero cannot be represented at all. This can be overcome by reserving the smallest exponent for this purpose, call it $e_{min} - 1$. Then floating-point zero is represented as $1.0 \times \beta^{e_{min}-1}$.

1.2.3 The numerical analyst's view of floating-point arithmetic

Ideally, in order to use classical mathematical techniques, we would like to ignore the use of floating-point arithmetic. However, the use of such 'approximate' arithmetic is relevant, although the numerical analyst does not want to be concerned with details of a particular implementation of floating-point. In this course we take a simplified view of machine arithmetic†. In general, arithmetic base and method of rounding are of no concern, except to note that some implementations are better than others. The floating-point precision is important. We will define the parameter *machine epsilon*, associated with the precision of each floating-point implementation, and then apply the same analysis for all implementations.

Before making this powerful simplification, we consider some implications of the use of floating-point arithmetic which will be swept under the carpet in so doing. These points often cause problems for the software designer rather than the numerical analyst:

(1) Floating-point is a finite arithmetic. This is not too serious, but is worth reflecting on. On most computers, for historical reasons, an integer and a floating-point number each occupy one word of storage, i.e. the same number of bits. Therefore there are the same number of representable numbers. For example, on an IBM mainframe a word is 32 bits so there are 2^{32} representable integers (between -2^{31} and $+2^{31}$) and 2^{32} representable floating-point numbers (between -2^{252} and $+2^{252}$).

† Except in Sections 1.7, 1.13 and 3.3.2 where floating-point arithmetic is discussed in detail.

(2) The representable floating-point numbers, under the arithmetic operations available on the computer, do not constitute a field‡. It is trivial to show this by multiplying two large numbers, the product of which is not representable (i.e. causes overflow). Less obvious deficiencies are more troublesome, e.g. in IBM single precision the number $X = -1.0 \times 10^{76}$ is representable but $-X$ is not.

(3) The representable floating-point numbers are more dense near zero. We shall see that this is reasonable, but leads to complications before we get very far.

1.2.4 Overflow and underflow

For convenience we will often use the terms *overflow* and *underflow* to denote numbers that are outside the range that is representable. It is worth pointing out that both overflow and underflow are hazards in numerical computation but in rather different ways.

We can regard overflow as being caused by any calculation whose result is too large in absolute value to be represented, e.g. as a result of exponentiation or multiplication or division or, just possibly, addition or subtraction. This is potentially a serious problem: if we cannot perform arithmetic with ∞ then we must treat overflow as an error. In a language with no *exception handling* there is little that can be done other than terminate the program with an error condition. In some cases overflow can be anticipated and avoided. In Section 1.13 we will see how IEEE arithmetic deals with overflow, and makes recovery possible in many cases.

Conversely, underflow is caused by any calculation whose result is too small to be distinguished from zero. Again this can be caused by several operations, although addition and subtraction are less likely to be responsible. However, we can perform ordinary arithmetic using zero (unlike ∞) so underflow is less of a problem but more insidious: often, but not always, it is safe to treat an underflowing value as zero. There are several exceptions. For example, suppose a calculation involving time uses a variable time step δt which is used to update the time t (probably in a loop terminated at some fixed time) by assignments of the form

$$\begin{aligned} delta\_t &:= delta\_t * some\_positive\_value \\ t &:= t + delta\_t \end{aligned}$$

If the variable *delta_t* ever underflows, then the calculation may go into an infinite loop. An 'ideal' algorithm would anticipate and avoid such problems.

It is also worth mentioning that overflow and underflow problems can often be overcome by re-scaling a problem to fit more comfortably into the representable number range.

1.3 Errors and machine epsilon

Suppose that x, y are real numbers not dangerously close to overflow or underflow. Let x^* denote the floating-point representation of x. We define the *absolute error* ε by

$$x^* = x + \varepsilon$$

and the *relative error* δ by

$$x^* = x(1+\delta) = x + x\delta$$

We can write

$$\varepsilon = x\delta$$

or, if $x \neq 0$,

$$\delta = \frac{\varepsilon}{x}.$$

‡ A field is a mathematical structure in which elements of a set A obey the formal rules of ordinary arithmetic with respect to a pair of operators representing addition and multiplication. The concepts of subtraction and division are implicit in these rules.

When discussing floating-point arithmetic, relative error seems appropriate because each number is represented to a similar relative accuracy. However, there is a problem when $x = 0$ or x is very close to 0 so we will need to consider absolute error as well.

When we write $x^* = x(1+\delta)$ it is clear that δ depends on x. For any floating-point implementation, there must be a number u such that $|\delta| \leq u$, for all x (excluding x values very close to overflow or underflow). The number u is usually called the *unit round off*. On most computers $1^* = 1$. The smallest positive ϵ such that

$$(1+\epsilon)^* > 1$$

is called *machine epsilon* or *macheps*. It is often assumed that $u \simeq macheps$.

Let ω represent any of the arithmetic operations $+ - */$ on real numbers. Let ω^* represent the equivalent floating-point operation. We naturally assume that

$$x\omega^* y \simeq x\omega y.$$

More specifically, we assume that

$$x\omega^* y = x\omega y(1+\delta) \tag{1.1}$$

for some δ ($|\delta| \leq u$). Equation (1.1) is very powerful and underlies *backward error analysis* which allows us to use ordinary arithmetic while considering the data to be 'perturbed'. That is to say, approximate arithmetic applied to correct data can be thought of as correct arithmetic applied to approximate data. The latter idea is easier to deal with algebraically and leads to less pessimistic analyses.

1.4 Error analysis

It is possible to illustrate the 'style' of error analyses by means of a very simple example. Consider the evaluation of a function $f(x) = x^2$. We would like to know how the error grows when a floating-point number is squared.

Forward error analysis tells us about worst cases. In line with this pessimistic view we will assume that all floating-point numbers are inaccurately represented with approximately the same relative error, but again assume that numbers are not close to overflow or underflow.

Forward error analysis

We express the relative error in x by

$$x^* = x(1+\delta).$$

Squaring both sides gives

$$\begin{aligned}(x^*)^2 &= x^2(1+\delta)^2 \\ &= x^2(1+2\delta+\delta^2) \\ &\simeq x^2(1+2\delta)\end{aligned}$$

since δ^2 is small, i.e. the relative error is approximately doubled.

Backward error analysis

To avoid confusion we now use ρ to denote the relative error in the result, i.e. we can write

$$[f(x)]^* = x^2(1+\rho) \tag{1.2}$$

such that $|\rho| \leq u$.† As ρ is small, $1+\rho > 0$ so there must be another number $\tilde{\rho}$ such that $(1+\tilde{\rho})^2 = 1+\rho$ where $|\tilde{\rho}| < |\rho| \leq u$. We can now write

$$\begin{aligned}[f(x)]^* &= x^2(1+\tilde{\rho})^2 \\ &= f\{x(1+\tilde{\rho})\}.\end{aligned}$$

We can interpret this result by saying that the error in squaring a floating-point number is no worse that the error in representing the result in floating-point form.

These results are dramatically different ways of looking at the same process. The forward error analysis tells us that squaring a number causes loss of accuracy. The backward error analysis suggests that this is not something we need to worry about, given that we have decided to use floating-point arithmetic in the first place. There is no contradiction between these results: they accurately describe the same computational process from different points of view.

Historically, forward error analysis was developed first and led to some very pessimistic predictions about how numerical algorithms would perform for large problems. When backward error analysis was developed in the 1950s by J.H. Wilkinson, the most notable success was in the solution of simultaneous linear equations by Gaussian elimination (see Section 2.3).

We will now investigate how errors can build up using the five basic floating-point operations: $+ - */ \uparrow$, where $\uparrow$ denotes exponentiation. Again it would be convenient to use ordinary arithmetic but consider the following problem: suppose numbers x and y are exactly represented in floating-point but that the result of computing $x * y$ is not exactly represented. We cannot explain where the error comes from without considering the properties of the particular implementation of floating-point arithmetic.

As an example, consider double precision arithmetic on an IBM 3084. The value of *macheps* is approximately 0.22×10^{-15}. For the sake of simplicity we will assume that all numbers are represented with the same relative error 10^{-15}.

We can deal most easily with multiplication. We write

$$\begin{aligned}x_1^* &= x_1(1+\delta_1) \\ x_2^* &= x_2(1+\delta_2)\end{aligned}$$

Then

$$\begin{aligned}x_1^* \times x_2^* &= x_1x_2(1+\delta_1)(1+\delta_2) \\ &= x_1x_2(1+\delta_1+\delta_2+\delta_1\delta_2)\end{aligned}$$

Ignoring $\delta_1\delta_2$ because it is small, the worst case is when δ_1 and δ_2 have the same sign, i.e. the relative error in $x_1^* \times x_2^*$ is no worse than $|\delta_1| + |\delta_2|$. Taking the IBM example, if we perform one million floating-point multiplications then at worst the relative error will have built up to $10^6.10^{-15} = 10^{-9}$.

We can easily dispose of division by using the binomial expansion to write

$$1/x_2^* = (1/x_2)(1+\delta_2)^{-1} = (1/x_2)(1-\delta_2+...).$$

Then, by a similar argument, the relative error in x_1^*/x_2^* is again no worse than $|\delta_1| + |\delta_2|$.

We can compute $x_1^* \uparrow n$, for any integer n by repeated multiplication or division. Consequently we can argue that the relative error in $x_1^* \uparrow n$ is no worse than $n|\delta_1|$.

This leaves addition and subtraction. Consider

$$\begin{aligned}x_1^* + x_2^* &= x_1(1+\delta_1) + x_2(1+\delta_2) \\ &= x_1 + x_2 + (x_1\delta_1 + x_2\delta_2) \\ &= x_1 + x_2 + (\varepsilon_1 + \varepsilon_2)\end{aligned}$$

† Equation (1.2) comes directly from (1.1) by setting $x = y$, $\delta = \rho$ and treating ω as 'multiply'.

where ε_1, ε_2 are the absolute errors in representing x_1, x_2 respectively. The worst case is when ε_1 and ε_2 have the same sign, i.e. the absolute error in $x_1^* + x_2^*$ is no worse than $|\varepsilon_1| + |\varepsilon_2|$.

Using the fact that $(-x_2)^* = -x_2 - \varepsilon_2$ we get that the absolute error in $x_1^* - x_2^*$ is also no worse that $|\varepsilon_1| + |\varepsilon_2|$.

The fact that error build-up in addition and subtraction depends on absolute accuracy, rather than relative accuracy, leads to a particular problem. Suppose we calculate $\sqrt{10} - \pi$ using a computer with *macheps* $\simeq 10^{-6}$, e.g. IBM single precision.

$$\begin{aligned} \sqrt{10} &= 3.16228 \\ \pi &= 3.14159 \\ \sqrt{10} - \pi &= 0.02089 \end{aligned}$$

Because these numbers happen to be of similar size the absolute error in representing each is around 3×10^{-6}. We expect that the absolute error in the result is about 6×10^{-6} at worst. But as

$$x_1^* - x_2^* = x_1 - x_2 + (\varepsilon_1 - \varepsilon_2)$$

the relative error in $x_1^* - x_2^*$ is

$$\frac{\varepsilon_1 - \varepsilon_2}{x_1 - x_2}$$

which, in this case, turns out to be about 3×10^{-4}. This means that the relative error in the subtraction is about 300 times as big as the relative error in x_1 or x_2.

This problem is known as *loss of significance*. It can occur whenever two similar numbers of equal sign are subtracted (or two similar numbers of opposite sign are added), and is a major cause of inaccuracy in floating-point algorithms. For example, if a calculation is being performed using, say, *macheps* $= 10^{-15}$ but one critical addition or subtraction causes a loss of significance with relative error, say, 10^{-10} then the relative error achieved may be only 10^{-5}.

However, loss of significance can be quite a subtle problem as the following two examples illustrate. Consider the function $\sin x$ which has the series expansion

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

which converges, for any x, to a value in the range $-1 \leq \sin x \leq 1$.

If we attempt to sum this series of terms with alternating signs using floating-point arithmetic we can anticipate that loss of significance will be a problem. Using a BBC micro (*macheps* $\simeq 2 \times 10^{-10}$) the following results may be obtained.

Example 1

Taking $x = 2.449$ the series begins

$$\sin x = 2.449 - 2.448020808 + 0.7341126024 - \ldots$$

and 'converges' after 11 terms to 0.6385346144, with a relative error barely larger than *macheps*. The obvious loss of significance after the first subtraction does not appear to matter.

Example 2

Taking $x = 20$ the series begins

$$\sin x = 20 - 1333.333333 + 26666.66667 - \ldots.$$

The $10th$ term, -43099804.09 is the largest and the $27th$ term is the first whose absolute value is less than 1. The series 'converges' after 37 terms to 0.9091075368 whereas $\sin 20 = 0.9129452507$, giving a relative error of about 4×10^{-3} despite the fact that no individual operation causes such a great loss of significance.

It is important to remember, in explaining these effects, that a sequence of addition/subtraction operations is being performed.

In Example 1 the evaluation of $2.449 - 2.448020808$ causes loss of significance, i.e. the relative error increases by at least 1000 times although the absolute error is at worst doubled. Adding in the next term, 0.7341126024, again makes little difference to the absolute error but, because the new term is much larger, the relative error is reduced again. Overall, the relative error is small.

In Example 2 the loss of significance is caused by the fact that the final result is small compared with numbers involved in the middle of the calculation. The absolute error in the result is the sum of the absolute error in each addition/subtraction. Calculations involving the largest term will contribute an absolute error of nearly 10^{-2} so we could estimate the final relative error to be approximately $10^{-2}/0.91$, which is slightly larger than the observed error.

Exercise 1a

In any language with floating-point arithmetic, write a program to evaluate

$$\sum_{j=1}^{n} x_j$$

(a) by summing over increasing values of j, and (b) by summing over decreasing values of j. Set $n = 100000$ and

$$x_1 = 1,$$
$$x_j = 1000\varepsilon/j, \quad j = 2, 3, \ldots n$$

where $\varepsilon \simeq macheps$. (You will need to discover, from documentation or otherwise, an approximate value of *macheps* for the precision that you are using.) Explain why the results of (a) and (b) are different.

1.5 Solving quadratics

Solving the quadratic equation $ax^2 + bx + c = 0$ appears, on the face of it, to be a very elementary problem. It is also a very important one, principally because it is often encountered as part of a larger calculation. Particularly important is the use of quadratic equations in matrix computations in which, typically, several thousand quadratics might need to be solved in a straightforward calculation.

Such applications require an algorithm for solution of a quadratic equation that is *robust* in the sense that it will not fail or give inaccurate answers for any reasonable representable coefficients a, b and c. We will now investigate how easy this is to achieve.

It is well known that the solution of $ax^2 + bx + c = 0$ can be expressed by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

A problem arises if $b^2 >> |4ac|$ in which case one root is small. Suppose $b > 0$ so that the small root is given by

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{1.3}$$

with loss of significance in the numerator. The problem can be averted by writing

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}}$$

and simplifying to get

$$x = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \tag{1.4}$$

so that the similar quantities to be summed are now of the same sign. Taking $a = 1$, $b = 100000$, $c = 1$ as an example and using a BBC micro again (*macheps* $\simeq 2 \times 10^{-10}$) we get

$$x = -1.525878906 \times 10^{-5} \qquad \text{using(1.3)}$$
$$x = -1.000000000 \times 10^{-5} \qquad \text{using(1.4)}$$

and the latter is as accurate as the machine precision will allow.

A robust algorithm must use equation (1.3) or (1.4) as appropriate in each case.

The solution of quadratic equations illustrates that even the simplest problems can present numerical difficulties although, if anticipated, the difficulties may be circumvented by adequate analysis.

Exercise 1b

The Bessel functions $J_0(x)$, $J_1(x)$, $J_2(x)$, ... satisfy the recurrence formula

$$J_{n+1}(x) = (2n/x)J_n(x) - J_{n-1}(x)$$

On a certain computer, when $x = 2/11$, the first two Bessel functions have the approximate values

$$J_0(x) = 0.991752$$
$$J_1(x) = 0.0905339$$

where there is known to be an error of about 0.5 in the last digit. Assuming $2/x$ evaluates to 11 exactly, and using six significant decimal digits, calculate $J_2(x)$ and $J_3(x)$ from the formula and estimate the approximate relative error in each. How accurately can $J_4(x)$ be calculated? (N.B. This exercise is designed so that the arithmetic is very simple, and a computer or calculator is unnecessary.)

When $x = 20$, using

$$J_0(x) = 0.167024$$
$$J_1(x) = 0.0668331$$

$J_4(x)$ can be evaluated to the same relative accuracy as $J_0(x)$ and $J_1(x)$. How do you account for this?

1.6 Convergence and error testing

An *iterative* numerical process is one in which a calculation is repeated to produce a sequence of approximate solutions. If the process is successful, the approximate solutions will *converge*.

Convergence of a sequence is usually defined as follows. Let x_0, x_1, x_2, ... be a sequence (of approximations) and let x be a number. We define

$$\varepsilon_n = x_n - x.$$

The sequence converges if

$$\lim_{n \to \infty} \varepsilon_n = 0$$

for some number x, which is called the *limit* of the sequence. The important point about this definition is that *convergence of a sequence* is defined in terms of *absolute error*. In a numerical algorithm we cannot really test for convergence as it is an infinite process, but we can check that the error is getting smaller, and consequently that convergence is likely. Writers of numerical software tend to talk loosely of 'convergence testing', by which they usually mean error testing. Some subtlety is required in error testing, as we will now see.

For simplicity, we now drop the suffix n.

Suppose that a numerical algorithm is being designed for a problem with a solution x. Let $\tilde{x}$ be an approximation to x, and let $\tilde{\varepsilon}$ be an estimate of the absolute error in x, i.e.

$$\tilde{x} \simeq x + \tilde{\varepsilon}.$$

Note that, for a typical problem, it is unlikely that we can find the *exact* absolute error, otherwise we would be able to calculate the *exact* solution from it.

Assume that a target absolute accuracy ε_t is specified. Then we could use a simple error test in which the calculation is terminated when

$$|\tilde{\varepsilon}| \leq \varepsilon_t. \tag{1.5}$$

Consider floating-point arithmetic with *macheps* $\simeq 10^{-16}$. If x is large, say 10^{20}, and $\varepsilon_t = 10^{-6}$ then $\tilde{\varepsilon}$ is never likely to be much less than 10^4, so condition (1.5) is unlikely to be satisfied even when the process converges.

Despite the definition of convergence it appears that relative error would be better, although it is unsafe to estimate relative error in case $\tilde{x} = 0$. Writing δ_t for target relative error, we could replace (1.5) with the error test

$$|\tilde{\varepsilon}| \leq \delta_t |\tilde{x}|. \tag{1.6}$$

In this case, if $|\tilde{x}|$ is very small then $\delta_t|\tilde{x}|$ may underflow‡ and then test (1.6) may never be satisfied (unless $\tilde{\varepsilon}$ is exactly zero).

As (1.5) is useful when (1.6) is not, and vice versa, so-called *mixed error tests* have been developed. In the simplest form of such a test, a target error η_t is prescribed and the calculation is terminated when the condition

$$|\tilde{\varepsilon}| \leq \eta_t(1 + |\tilde{x}|) \tag{1.7}$$

is satisfied. If $|\tilde{x}|$ is small η_t may be thought of as target absolute error, or if $|\tilde{x}|$ is large η_t may be thought of as target relative error.

A test like (1.7) is used in much modern numerical software, but it solves only part of the problem. We also need to consider how to estimate ε. The simplest formula is

$$\tilde{\varepsilon}_n = x_n - x_{n-1} \tag{1.8}$$

where we again use the subscript n to denote the *nth* approximation. Suppose an iterative method is used

‡ Note that the treatment of underflowed values may depend on both the floating-point architecture and the programming language in use.

to find the positive root of the equation $x^4 = 16$ with successive approximations

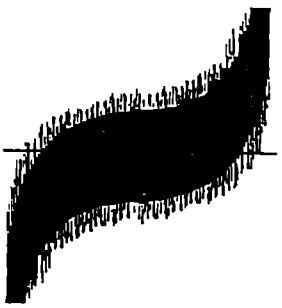
$$x_0 = 4$$
$$x_1 = 0.5$$
$$x_2 = 2.25$$
$$x_3 = 2.25$$
$$x_4 = 1.995$$
$$x_5 = 2.00001$$
$$x_6 = 1.999999998$$

If (1.8) is used then the test (1.7) will cause premature termination of the algorithm with the incorrect answer 2.25. In case this contrived example is thought to be unlikely to occur in practice, theoretical research has shown that such cases must always arise for a wide class of numerical methods. A safer test is given by

$$\bar{\varepsilon}_n = |x_n - x_{n-1}| + |x_{n-1} - x_{n-2}| \tag{1.9}$$

but again research has shown that x_{n-2}, x_{n-1} and x_n can all coincide for certain methods so (1.9) is not guaranteed to work. In many problems, however, confirmation of convergence can be obtained independently: e.g. in the example above it can be verified by computation that 2.25 does not satisfy the equation $x^4 = 16$.

Another important problem associated with 'convergence detection' is that the 'granularity' of a floating-point representation leads to some uncertainty† as to the exact location of, say, a zero of a function $f(x)$. Suppose that the mathematical function $f(x)$ is smooth and crosses the x axis at some point a. Suppose also that $f(x)$ is evaluated on a particular computer for every representable number in a small interval around $x = a$. The graph of these values, when drawn at a suitably magnified scale might appear as follows.



If we state that $f(a) = 0$ then there is uncertainty as to the value of a. Alternatively, if we fix a point a, then there is uncertainty in the value of $f(a)$ unless a has an exact representation in floating-point. Note particularly, that a 1-bit change in the representation of x might produce a change of sign in the representation of $f(x)$.

This problem is easily avoided by specifying a target error that is not too close to *macheps*.

1.7 Rounding error in floating-point arithmetic

The term *infinite precision* is used below to denote the result of a basic floating point operation $(+ - */)$ as if performed exactly and then rounded to the available precision. Note that a calculation can actually be performed in this way for the operations $+ -$ and $*$, but not for / as the result of a division may require

† Physicists may note that there is an analogy here with quantum physics and uncertainty principles.

infinitely many digits to store. However, the end result of infinite precision division can be computed by a finite algorithm.

Suppose $\beta = 10$ and $p = 3$. If the result of a floating-point calculation is 3.12×10^{-2} when the result to infinite precision is 3.14×10^{-2}, then there is said to be an error of 2 *units in the last place*. The acronym *ulp* (plural *ulps*) is often used for this. When comparing a real number with a machine representation, it is usual to refer to fractions of an *ulp*. For example, if the number π (whose value is the non-terminating non-repeating decimal 3.1415926535...) is represented by the floating-point number 3.14×10^0 then the error may be described as 0.15926535... *ulps*. The maximum error in representing any accurately known number (except near to underflow or overflow) is 1/2 *ulp*.

The numerical analyst does not want to be bothered with concepts like '*ulp*' because the error in a calculation, when measured in *ulps*, varies in a rather haphazard way and is different from machine range to machine range. The numerical analyst is more interested in *relative error*. An interesting question is: how does the relative error in an operation vary between different floating-point implementations? This problem is 'swept under the carpet' in most numerical analysis. Subtraction of similar quantities is a notorious source of error. If a floating-point format has parameters β and p, and subtraction is performed using p digits, then the worst relative error of the result is $\beta - 1$, i.e. base 16 can be 15 times less accurate than base 2. This occurs, for example, in the calculation

$$1.000\ldots0 \times \beta^e - 0.\rho\rho\rho\ldots\rho \times \beta^e$$

where $\rho = \beta - 1$ if the least significant ρ is lost during the subtraction. This property implies that the smaller the base the better, and that $\beta = 2$ is therefore optimal in this respect. Furthermore base 16, favoured by IBM, is a rather disadvantageous choice.

To see how this works in practice, compare the two representations $\beta = 16$, $p = 1$ and $\beta = 2$, $p = 4$. Both of these require 4 bits of significand. The worst case for hexadecimal is illustrated by the representation of 15/8. In binary, 15 is represented by 1.111×2^3 so that 15/8 is 1.111×2^0. In hexadecimal the digits are normally written as

$$0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,A\,B\,C\,D\,E\,F$$

so the number 15 is represented simply by $F \times 16^0$ whereas 15/8 can be represented no more accurately than 2×16^0. Although this example is based on 4-bit precision, it is generally true that 3 bits of accuracy can be lost in representing a number‡ in base 16.

A relatively cheap way of improving the accuracy of floating-point arithmetic is to make use of extra digits, called *guard digits*, in the computer's arithmetic unit only. If x and y are numbers represented exactly on a computer and $x - y$ is evaluated using just one guard digit, then the relative error in the result is less than $2 \times$ *macheps*. Although this only applies if x and y are represented exactly, this property can be used to improve expression evaluation algorithms.

In this Section the rather loosely defined term *mathematically equivalent* is used to refer to different expressions or formulae that are equivalent in true real arithmetic.

For example, consider the evaluation of $x^2 - y^2$ when $x \simeq y$, given exactly known x and y. If x^2 or y^2 is inexact when evaluated then loss of significance will usually occur when the subtraction is performed. However, if the (mathematically equivalent) calculation $(x + y)(x - y)$ is performed instead then the relative error in evaluating $x - y$ is less than $2 \times$ *macheps* provided there is at least one guard digit, because x and y are exactly known. Therefore serious loss of significance will not occur in the latter case.

Careful consideration of the properties of floating-point arithmetic can lead to algorithms that make little sense in terms of real arithmetic. Consider the evaluation of $\ln(1+x)$ for x small and positive. Suppose there exists a function $LN(y)$ that returns an answer accurate to less than 1/2 *ulp* if y is accurately represented. If $x <$ *macheps* then $1 + x$ evaluates to 1 and $LN(1)$ will presumably return 0. A more accurate approximation

‡ There is a compensating advantage in using base 16: the exponent range is greater, but this is not really as useful as increased accuracy.

in this case is $\ln(1+x) \simeq x$. However there is also a problem if $x > macheps$ but x is still small, because several digits of x will be lost when it is added to 1. In this case, $\ln(1+x) \simeq LN(1+x)$ is less accurate than the (mathematically equivalent) formula

$$\ln(1+x) \simeq x * LN(1+x)/((1+x)-1).$$

Indeed it can be proved that the relative error is at most $5 \times macheps$ for $x < 0.75$, provided at least one guard digit is used†.

The cost of a guard digit is not high in terms of processor speed: typically 2% per guard digit for a double precision adder. However the use of guard digits is not universal, e.g. Cray supercomputers do not have them.

The idea of *rounding* is fairly straightforward. Consider base 10, as it is familiar, and assume $p = 3$. The number 1.234 should be rounded to 1.23 and 1.238 should be rounded to 1.24. Most computers perform rounding, although it is only a user-specifiable option on some machine ranges. Curiously, IBM mainframes have no rounding and merely chop off extra digits.

The only point of contention is what to do about rounding in half-way cases: should 1.235 be rounded to 1.23 or to 1.24? A common method is to specify that the digit 5 rounds up, on the grounds that half of the digits round down and the other half round up. This is used, for example, on DEC VAX computers. Unfortunately this introduces a slight bias.

An alternative is to use 'round to even': the number 1.235 rounds to 1.24 because the digit 4 is even, whereas 1.265 rounds to 1.26 for the same reason. The slight bias of the above method is removed. Reiser & Knuth (1975) highlighted this by considering the sequence

$$x_n = (x_{n-1} - y) + y$$

where $n = 0, 1, 2, \ldots$ and the calculations are performed in floating point arithmetic. Using 'round to even' they showed that $x_n = x_1$ for all n, for any starting values x_0 and y. This is not true of the '5 rounds up' method. For example, let $\beta = 10$, $p = 3$ and choose $x_0 = 1.00$ and $y = -5.55 \times 10^{-1}$. This leads to the sequence $x_1 = 1.01$, $x_2 = 1.02$, $x_3 = 1.03$, ... with 1 *ulp* being added at each stage. Using 'round to even', $x_n = 1.00$ for all n.

It is also worth mentioning that probabilistic rounding has been suggested, i.e. rounding half-way cases up or down at random. This is also unbiased and has the advantage of reducing the overall standard deviation of errors, so could be considered more accurate, but it has the disadvantage that repeatability of calculations is lost. It is not implemented on any modern hardware.

Finally, it is often argued that many of the disadvantages of floating-point arithmetic can be overcome by using arbitrary (or variable) precision rather than a fixed precision. While there is some truth in this claim, efficiency is important and arbitrary precision algorithms are very often prohibitively expensive. This subject is not discussed here.

Exercise 1c

Explain the term *infinite precision*. Consider a floating-point implementation with $\beta = 10$, $p = 2$. What are the infinite precision results of the calculations

(a) $(1.2 \times 10^1) * (1.2 \times 10^1)$ (b) $(2.0 \times 10^0)/(3.0 \times 10^0)$

How many decimal guard digits are required in the significand of the arithmetic unit to calculate (b) to infinite precision?

† See Goldberg's paper for an explanation of how this works.

1.8 Norms

Up to now we have discussed only scalar problems, i.e. problems with a single number x as solution. The more important practical numerical problems often have a vector $\mathbf{x} = \{x_1, x_2, ...x_n\}$ of solutions‡. Although this is a major complication, it is surprising how often a numerical method for an n-dimensional problem can be derived from a scalar method simply by making the appropriate generalisation in the algebra. When programming a numerical algorithm for a vector problem it is still desirable to use an error test like (1.7) in the last section, but it is necessary to generalise this by introducing a measurement of the size of a vector.

In order to compare the size of vectors we need a scalar number, called the *norm*, which has properties appropriate to the concept of 'size':

(1) The norm should be positive, unless the vector is $\{0, 0, ...0\}$ in which case it should be 0.

(2) If the elements of the vector are made systematically smaller or larger then so should the norm.

(3) If two vectors are added or subtracted then the norm of the result should be suitably related to the norms of the original vectors.

The *norm* of a vector $\mathbf{x}$ is usually written $||\mathbf{x}||$ and is any quantity satisfying the three axioms:

(1) $||\mathbf{x}|| \geq 0$ for any $\mathbf{x}$. $||\mathbf{x}|| = 0 \iff \mathbf{x} = \mathbf{0} = \{0, 0, \ldots 0\}$.

(2) $||c\mathbf{x}|| = |c|.||\mathbf{x}||$ where c is any scalar.

(3) $||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}||$ where $\mathbf{x}$ and $\mathbf{y}$ are any two vectors.

The norms most commonly used are the so-called l_p norms in which, for any real number $p \geq 1$,

$$||\mathbf{x}||_p = \left\{ \sum_{i=1}^{n} |x_i|^p \right\}^{\frac{1}{p}}.$$

In practice, only the cases $p = 1$ or 2 or $\lim_{p \to \infty} ||\mathbf{x}||_p$ are used. The relevant formulae are:

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i| \qquad l_1 norm$$

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \qquad l_2 norm$$

$$||\mathbf{x}||_\infty = \max_i |x_i| \qquad l_\infty norm$$

Alternative names for the l_2 norm are: Euclidean norm, Euclidean length, length, mean. Alternative names for the l_∞ norm are: max norm, uniform norm, Chebyshev norm.

Consider the following example from an iterative process with a vector of solutions:

$$\mathbf{x_1} = \{1.04, 2.13, 2.92, -1.10\}$$
$$\mathbf{x_2} = \{1.05, 2.03, 3.04, -1.04\}$$
$$\mathbf{x_3} = \{1.01, 2.00, 2.99, -1.00\}.$$

This appears to be converging to something like $\{1, 2, 3, -1\}$, but we need to be able to test for this in a program. If we use, for example, a vector generalisation of (1.8), i.e. $\tilde{\mathbf{e}}_\mathbf{n} = \mathbf{x_n} - \mathbf{x_{n-1}}$ then we get

$$\tilde{\mathbf{e}}_\mathbf{2} = \{0.01, -0.10, 0.12, 0.06\}$$
$$\tilde{\mathbf{e}}_\mathbf{3} = \{-0.04, -0.03, -0.05, 0.04\}.$$

‡ Note that x_n does not have the same meaning as in the last section.

The norms of these error vectors are as follows

	l_1	l_2	l_∞
$\bar{e}_2$	0.29	0.17	0.12
$\bar{e}_3$	0.16	0.08	0.05

Note that in each case $||\bar{e}_3|| < ||\bar{e}_2||$ so it does not matter, for 'convergence testing' purposes, which norm is used.

A simple vector generalisation of the error test (1.7) is

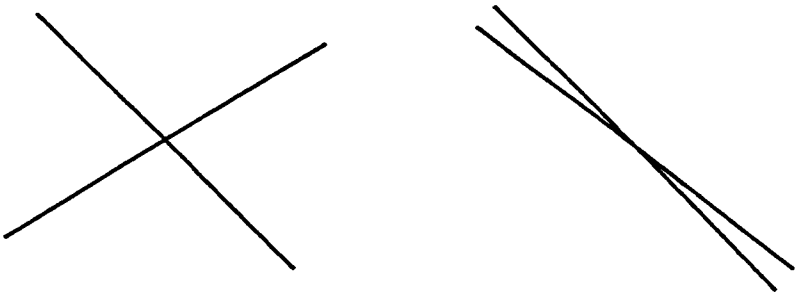
$$||\bar{e}|| \leq \eta_t(1 + ||\bar{x}||)$$

where η_t is the prescribed (scalar) error and any suitable norm is used. (Of course, the same norm must be used for both $||\bar{e}||$ and $||\bar{x}||$ so that they can be compared.)

1.9 Condition of a problem

The *condition* of a numerical problem is a qualitative or quantitative statement about how easy it is to solve, irrespective of the algorithm used to solve it.

As a qualitative example, consider the solution of two simultaneous linear equations. The problem may be described graphically by the pair of straight lines representing each equation: the solution is then the point of intersection of the lines. (The method of solving a pair of equations by actually drawing the lines with ruler and pencil on graph paper and measuring the coordinates of the solution point may be regarded as an algorithm that can be used.) Two typical cases are illustrated below:



well conditioned ill conditioned

The left-hand problem is easier to solve than the right hand one, irrespective of the graphical algorithm used. For example, a better (or worse) algorithm is to use a sharper (or blunter) pencil: but in any case it should be possible to measure the coordinates of the solution more exactly in the left-hand case than the right.

Quantitatively, the *condition* K of a problem is a measure of the sensitivity of the problem to a small perturbation. For example (and without much loss of generality) we can consider the problem of evaluating a differentiable function $f(x)$. Let x^* be a point close to x. In this case K is a function of x defined as the relative change in $f(x)$ caused by a *unit relative change* in x. That is

$$\begin{aligned} K(x) &= \lim_{x^* \to x} \frac{|[f(x) - f(x^*)]/f(x)|}{|(x - x^*)/x|} \\ &= \left|\frac{x}{f(x)}\right| \lim_{x^* \to x} \frac{|f(x) - f(x^*)|}{|x - x^*|} \\ &= \left|\frac{x.f'(x)}{f(x)}\right| \end{aligned}$$

from the definition of $f'(x)$.

Example 3

Suppose $f(x) = \sqrt{x}$. We get

$$K(x) = \left| \frac{x.f'(x)}{f(x)} \right| = \frac{x.[\frac{1}{2}/\sqrt{x}]}{\sqrt{x}} = \frac{1}{2}.$$

So K is a constant which implies that taking square roots is equally well conditioned whatever the value of x, and that the relative error is reduced by half in the process.

Example 4

Suppose now that $f(x) = \frac{1}{1-x}$. In this case we get

$$K(x) = \left| \frac{x.f'(x)}{f(x)} \right| = \left| \frac{x[1/(1-x)^2]}{1/(1-x)} \right| = \left| \frac{x}{1-x} \right|.$$

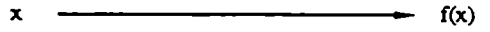
This $K(x)$ can get arbitrarily large for values of x close to 1 and can be used to estimate the relative error in $f(x)$ for such values, e.g. if $x = 1.000001$ then the relative error will increase by a factor of about 10^6.

Exercise 1d

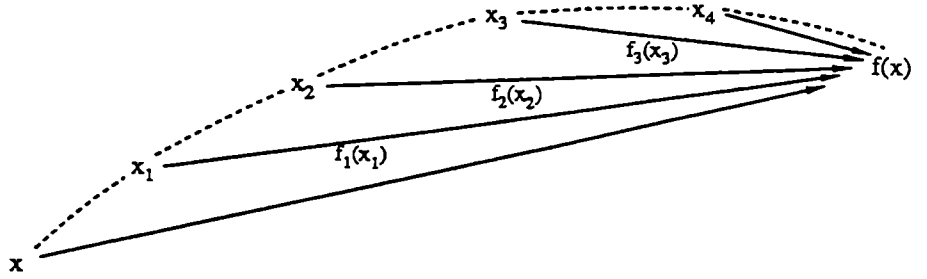
Examine the condition of the problem of evaluating $\cos x$.

1.10 Stability of an algorithm

If we represent the evaluation of a function by the rather informal graph



then we can represent an algorithm as a sequence of functions



where $x = x_0$ and $x_n = f(x)$. An algorithm is a particular method for solving a given problem, in this case the evaluation of a function.

Alternatively, an algorithm can be thought of as a sequence of problems, i.e. a sequence of function evaluations. In this case we consider the algorithm for evaluating $f(x)$ to consist of the evaluation of the sequence $x_1, x_2, \ldots x_n$. We are concerned with the condition of each of the functions $f_1(x_1)$, $f_2(x_2)$, $\ldots f_{n-1}(x_{n-1})$ where $f(x) = f_i(x_i)$ for all i.

An algorithm is *unstable* if any f_i is ill-conditioned, i.e. if any $f_i(x_i)$ has condition much worse than $f(x)$. Consider the example

$$f(x) = \sqrt{x+1} - \sqrt{x}$$

so that there is potential loss of significance when x is large. Taking $x = 12345$ as an example, one possible algorithm is

$$\begin{aligned} x_0 &:= x = 12345 \\ x_1 &:= x_0 + 1 \\ x_2 &:= \sqrt{x_1} \\ x_3 &:= \sqrt{x_0} \\ f(x) := x_4 &:= x_2 - x_3 \end{aligned} \tag{1.10}$$

The loss of significance occurs with the final subtraction. We can rewrite the last step in the form $f_3(x_3) = x_2 - x_3$ to show how the final answer depends on x_3. As $f_3'(x_3) = -1$ we have the condition

$$K(x_3) = \left| \frac{x_3 f_3'(x_3)}{f_3(x_3)} \right| = \left| \frac{x_3}{x_2 - x_3} \right|$$

from which we find $K(x_3) \simeq 2.2 \times 10^4$ when $x = 12345$. Note that this is the *condition* of a *subproblem* arrived at during the algorithm.

To find an alternative algorithm we write

$$\begin{aligned} f(x) &= (\sqrt{x+1} - \sqrt{x}) \frac{\sqrt{x+1} + \sqrt{x}}{\sqrt{x+1} + \sqrt{x}} \\ &= \frac{1}{\sqrt{x+1} + \sqrt{x}} \end{aligned}$$

This suggests the algorithm

$$\begin{aligned} x_0 &:= x = 12345 \\ x_1 &:= x_0 + 1 \\ x_2 &:= \sqrt{x_1} \\ x_3 &:= \sqrt{x_0} \\ x_4 &:= x_2 + x_3 \\ f(x) := x_5 &:= 1/x_4 \end{aligned} \tag{1.11}$$

In this case $f_3(x_3) = 1/(x_2 + x_3)$ giving a condition for the subproblem of

$$K(x_3) = \left| \frac{x_3 f_3'(x_3)}{f_3(x_3)} \right| = \left| \frac{x_3}{x_2 + x_3} \right|$$

which is approximately 0.5 when $x = 12345$, and indeed in any case where x is much larger than 1.

Thus algorithm (1.10) is unstable and (1.11) is stable for large values of x. In general such analyses are not usually so straightforward but, in principle, stability can be analysed by examining the condition of a sequence of subproblems.

Exercise 1e

Suppose that a function ln is available to compute the natural logarithm of its argument. Consider the calculation of $\ln(1 + x)$, for small x, by the following algorithm

$$\begin{aligned} x_0 &:= x \\ x_1 &:= x_0 + 1 \\ f(x) := x_2 &:= \ln(x_1) \end{aligned}$$

By considering the condition $K(x_1)$ of the subproblem of evaluating $\ln(x_1)$, show that such a function ln is inadequate for calculating $\ln(1 + x)$ accurately.

1.11 Order of convergence

Let $x_0, x_1, x_2, \ldots$ be a sequence of successive approximations in the solution of a numerical problem. We define the absolute error in the nth approximation by

$$x_n = x + \varepsilon_n.$$

If there exist constants p and C such that

$$\lim_{n\to\infty} \left| \frac{\varepsilon_{n+1}}{\varepsilon_n^p} \right| = C$$

then the process has *order of convergence* p, where $p \geq 1$. This is often expressed as

$$|\varepsilon_{n+1}| = O(|\varepsilon_n|^p)$$

using the O-notation. We can say that order p convergence implies that

$$|\varepsilon_{n+1}| \simeq C|\varepsilon_n|^p$$

for sufficiently large n. Obviously, for $p = 1$ it is required that $C < 1$, but this is not necessary for $p > 1$.

Various rates of convergence are encountered in numerical algorithms, the most important of which are as follows.

$p = 1$: linear convergence. Each interation produces the same reduction in absolute error. This is generally regarded as being too slow for practical methods.

$p = 2$: quadratic convergence. Each iteration squares the absolute error, which is very satisfactory provided the error is small. This is sometimes regarded as the 'ideal' rate of convergence.

$1 < p < 2$: superlinear convergence. This is not as good as quadratic but, as the reduction in error increases with each interation, it may be regarded as the minimum acceptable rate for a useful algorithm.

$p > 2$. Such methods are unusual, mainly because they are restricted to very smooth functions and often require considerably more computation than quadratic methods.

Exponential rate of convergence. This is an interesting special case.

It is worth mentioning a technique known as *convergence acceleration*. This is often used to improve linear convergence to a quadratic rate or better. It is useful when the convergence acceleration method itself requires less computation than using a higher order method. Such techniques are Aitken's δ^2 process and Wynn's ε algorithm which are beyond the scope of this course.

1.12 Computational complexity

The ideal algorithm should not only be stable, and have a fast rate of convergence, but should also have a reasonable *computational complexity*. It is possible to devise a stable algorithm that has, say, a quadratic rate of convergence but is still too slow to be practical for large problems. Suppose that a problem involves computations on a matrix of size $N \times N$. If the computation time increases rapidly as N increases then the algorithm will be unsuitable for calculations with large matrices.

Suppose that some operation, call it $\odot$, is the most expensive in a particular algorithm. If the time spent by the algorithm may be expressed as $O[f(N)]$ operations of type $\odot$, then we say that the *computational complexity* is $f(N)$.

In matrix calculations, the most expensive operations are multiplication and array references. For this purpose the operation $\odot$ may be taken to be the combination of a multiplication and one or more array

references. Useful matrix algorithms typically have computational complexity N^2, N^3 or N^4. These effectively put limits on the sizes of matrices that can be dealt with in a reasonable time. As an example, consider multiplication of matrices $\mathbf{A} = (a_{ij})$ and $\mathbf{B} = (b_{ij})$ to form a product $\mathbf{C} = (c_{ij})$. This requires N^2 sums of the form

$$c_{ij} = \sum_{k=1}^{N} a_{ik}.b_{kj}$$

each of which requires N multiplications (plus array references). Therefore the computational complexity is $N^2.N = N^3$.

Note that operations of lower complexity do not change the overall computational complexity of an algorithm. For example, if an N^2 process is performed each time an N^3 process is performed then, because

$$O(N^2) + O(N^3) = O(N^3)$$

the overall computational complexity is still N^3.‡

1.13 The IEEE Floating-point Standards

There are in fact two IEEE Standards, but in this section it is only occasionally necessary to distinguish between them as they are based on the same principles.

The Standard known as IEEE 754 requires that $\beta = 2$ and specifies the precise layout of bits in both *single precision* ($p = 24$) and *double precision* ($p = 53$).

The other Standard, IEEE 854, is more general and allows either $\beta = 2$ or $\beta = 10$. The values of p for single and double precision are not exactly specified, but constraints are placed on allowable values for them. Base 10 is included mainly for calculators, for which there is some advantage in using the same number base for the calculations as for the display. However, bases 8 and 16 are not supported by the Standard.

IEEE arithmetic is implemented by several manufacturers, e.g. Sun, but not yet by all, e.g. IBM System/370 mainframes.

If binary is used, one extra bit of significance can be gained because the first bit of the significand of a normalized number must always be 1, so need not be stored. Such implementations are said to have a *hidden bit*. Note that this device does not work for any other base.

IEEE 754 single precision uses 1 bit for the sign (of the significand), 8 bits for the exponent, and 23 bits for the significand. However, $p = 24$ because a hidden bit is used. Zero is represented by an exponent of $e_{\min} - 1$ and a significand of all zeros. There are other special cases also, as discussed below. Single precision numbers are designed to fit into 32 bits and double precision numbers, which have both a higher precision† and an extended exponent range, are designed to fit into 64 bits. IEEE 754 also defines two types of 'extended precision', but these are only specified in terms of minimum requirements: these are intended for cases where a little extra precision is essential in a calculation. The table below summarises the four precisions.

‡ If, however, the N^2 process was performed N^2 times each time the N^3 process was performed then the computational complexity would be $N^2.N^2 = N^4$.

† Note that *double precision* has more than double the precision of *single precision*.

	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
$e_{\max}$	+127	$\geq +1023$	+1023	$\geq +16383$
$e_{\min}$	−126	≤ -1022	−1022	≤ -16382
exponent width (bits)	8	≥ 11	11	≥ 15
total width (bits)	32	≥ 43	64	≥ 79

Note that an implementation which has hardware double precision would probably use this whenever 'single extended' was appropriate, to avoid implementing all four precisions. Also note that 'double extended' is sometimes referred to as '80-bit format', even though it only requires 79 bits. This is for the extremely practical reason that 'double extended' is often implemented by software emulation, rather than hardware, so it is safer to assume that the 'hidden bit' might not be hidden after all!

The IEEE Standard uses 1 bit for the sign of the significand and the remaining bits for its magnitude‡.

Exponents are stored using a *biased representation.* If e is the exponent of a floating-point number, then the bit-pattern of the stored exponent has the value $e + e_{\max}$ when it is interpreted as an unsigned integer.

Under IEEE, the operations $+ \ - \ * \ /$ must all be performed accurately, i.e. as in infinite precision, rounded to the nearest correct result, using 'round to even'. This can be implemented efficiently, but requires a minimum of two guard digits plus one extra bit, i.e. three guard digits in the binary case.

The IEEE Standard specifies that the square root and remainder operations must be correctly rounded, and also conversions between integer and floating-point types. The conversion between internal floating-point format and decimal for display purposes must be done accurately, apart from numbers close to overflow.

The IEEE Standard does not specify that transcendental functions be exactly rounded for two reasons: (1) correct implementation of 'round to even' can in principle require an arbitrarily large amount of computation, and (2) no algorithmic definitions were found to be suitable across all machine ranges†.

It has been pointed out that the IEEE Standard has failed to specify how inner products of the form

$$\sum_{i=1}^{n} x_i . y_i$$

are calculated, as incorrect answers can result if extra precision is not used. It is feasible to standardise the calculation of inner products, and it is to be hoped that this can be added to the Standard at some time in the future.

Some older floating-point representations, e.g. that used on IBM System/370 mainframes, treat every bit pattern as a valid floating-point number. In some programming languages, e.g. Standard Fortran 77, there is no exception handling, so there is little prospect of recovering from certain kinds of numerical error. For example, using strictly Standard Fortran 77 on the above computer, and real arithmetic, calling the square root function with the argument −4 is catastrophic. An error message is printed and the program stops; an option does allow the program to continue after such an error, and the square root function then has to return some valid floating-point result — but none of them is correct!

‡ The 'sign/magnitude' method was preferred to the alternative '2s-complement' method in which the significand is represented by the smallest non-negative number that is congruent to it modulo 2^p.

† The modern trend is towards large tables from which values of the common transcendental functions can be interpolated, but the storage overhead makes this method unsuitable for small computers.

The IEEE Standard attempts to tackle this problem without assuming that exception handling is available. Special values (all with exponents $e_{\max}+1$ or $e_{\min}-1$) are reserved for the special quantities ± 0, $\pm\infty$, denormalized numbers, and the concept of NaN ($Not\ a\ Number$). There are two (signed) values of zero, although IEEE is at pains to specify that they are indistinguishable in normal arithmetic. Operations that overflow yield $\pm\infty$, and operations that underflow yield ± 0. Note that other calculations can yield $\pm\infty$ (e.g. $x/0$) or ± 0 (e.g. x/∞).

There are many $NaNs$, though the significance of the different values is not standardised. To all intents and purposes, the user can regard all $NaNs$ as identical, although system-dependent information may be contained in a particular NaN value. The table below shows how the various categories of number are stored.

Exponent	Fraction	Represents	
$e = e_{\min}-1$	$f = 0$	± 0	zero
$e = e_{\min}-1$	$f \neq 0$	$\pm 0.f \times 2^{e_{\min}}$	denormalized numbers
$e_{\min} \leq e \leq e_{\max}$	any f	$\pm 1.f \times 2^{e}$	normalized numbers
$e = e_{\max}+1$	$f = 0$	$\pm\infty$	infinities
$e = e_{\max}+1$	$f \neq 0$	NaN	$Not\ a\ Number$

Note that as a special exponent is used to denote denormalized numbers, the 'hidden bit' device can be used for these as well, except that the hidden bit is always 0 in this case. The following is a list of operations that produce a NaN:

any operation involving a NaN	∞/∞
$\infty + (-\infty)$	x REM 0
$0 * \infty$	∞ REM x
$0/0$	$\sqrt{x}$ for $x < 0$

A manufacturer may choose to use the significand of a NaN to store system specific information. If so then the result of any operation between a NaN and a non-NaN must be the value of the NaN; the result of an operation between two $NaNs$ must be the value of one of them.

An arithmetic operation involving $\pm\infty$ typically produces a NaN or $\pm\infty$, where the sign of ∞ is determined by the usual rules of arithmetic. An exception is that $\pm x/\pm\infty$ is defined to be ± 0 for any ordinary number x. This has advantages and disadvantages. For example, consider the evaluation of $f(x) = x/(x^2+1)$. If x is so large that x^2 evaluates to ∞ then $f(x)$ evaluates to 0 instead of the (representable) approximation $1/x$. This can be turned to advantage by evaluating $f(x) = 1/(x+1/x)$ instead; not only does the above problem go away, but $f(0)$ is correctly evaluated without having to treat it as a special case‡. The only real disadvantage of 'infinity arithmetic' is that poorly constructed algorithms can produce wrong results, whereas they would produce error messages or raise exceptions on non-IEEE machines.

The fact that IEEE arithmetic has two representations of zero is controversial. One justification for it is that $1/(1/x)$ evaluates to x for all x. Another is that two signed zeros are sometimes useful to refer to function values on either side of a discontinuity; this is particularly useful in complex arithmetic but is not discussed

‡ In general, reducing the number of tests for special cases improves efficiency on pipelined machines or when optimizing compilers are used.

here. A simple real arithmetic example is that $\log(+0)$ can be defined as $-\infty$ and $\log(-0)$ as a NaN to distinguish between underflowed cases.

The IEEE Standard compromises by requiring that $+0 = -0$ in all tests. In particular the test '**if** $(x = 0)$ **then** ...' does not take the sign of x into account. However the sign of 0 is preserved in operations to the extent that, say, $(-3) * (-0)$ evaluates to $+0$ and $(+0)/(-3)$ evaluates to -0.

The IEEE Standard includes denormal numbers mainly to ensure that the property

$$x = y \quad \text{if and only if} \quad x - y = 0$$

holds† for all finite x. For this to be true, denormal numbers are required to represent $x - y$ when its value would otherwise underflow to ± 0. Also program code such as

$$\textbf{if} \ \ (x \neq y) \ \ \textbf{then} \ \ z = 1/(x - y)$$

could otherwise fail due to division by zero. This self-consistency is an elegant feature of IEEE arithmetic.

The IEEE Standard includes the concept of *status flags* which are set when various exceptions occur. Implementations are required to provide a means to read and write these from programs. Once set, a status flag will remain so until explicitly cleared. One use of these is to distinguish between an ordinary overflow and a calculation that genuinely yields ∞, e.g. $1/0$.

The IEEE Standard allows for *trap handlers* that handle exceptions, and read and write status flags. However trap handlers, which are 'procedures', are merely recommended and not required by the Standard.

There are five status flags corresponding to the exceptions: overflow, underflow, division by zero, invalid operation, and inexact. The first three are straightforward. An *invalid operation* is any operation that gives rise to a NaN when none of its operands is a NaN. An *inexact* exception arises simply when an operation cannot be evaluated exactly in floating-point, which is clearly a very common occurrence.

It is recommended that the status flags be implemented by software for efficiency. For example, the *inexact* exception will occur very frequently so it is more efficient for software to disable hardware testing for this once it has occurred. If the appropriate status flag is reset then testing for the *inexact* exception can be re-enabled.

There are various useful applications of trap handlers. For example, for computing

$$\prod_{i=1}^{n} x_i$$

where the product may overflow or underflow at an intermediate stage, even if the result is within the representable range. The evaluation of this product can be implemented by means of overflow and underflow trap handlers. The overflow trap handler works by maintaining a count of overflows and *wrapping* the exponent of a product so that it remains within the representable range‡. The underflow handler does the converse. At the end of the calculation the result is within the representable range if the number of overflows is equal to the number of underflows; in this case the wrapping algorithm ensures that the final exponent is correct. The cost is almost nothing if no intermediate product overflows or underflows.

In order that this sort of algorithm can be easily implemented, IEEE 754 specifies that the overflow and underflow handlers must be passed the offending value, as an argument, in 'wrapped-around' form.

---

† Goldberg notes that properties such as this, that are useful to make programs provable, tend to lead to better programming practice, even though proving large programs correct may be impractical.

‡ Wrapping, in this sense, means using modular arithmetic to ensure that any exponent is representable. If counts are kept of overflows and underflows then the true exponent can be recovered.

By default, IEEE arithmetic rounds to the nearest number using 'round to even'. Three other alternatives are provided: round towards 0, round towards $-\infty$, and round towards $+\infty$. A combination of the latter two enables 'interval arithmetic' to be performed.

The rationale for the 'extended' precisions is that a floating-point algorithm often requires some extra precision for certain operations; however modern computer instruction sets tend not to provide this facility. The multiplication of two numbers to produce a result of greater precision is an operation that is particularly useful. Specifically, for calculating (1) $b^2 - 4ac$ in the quadratic formula, (2) inner products, and (3) the correction to an approximation in certain iterative algorithms.

Exercise 1f

What are the results of the following operations?

$$1/\infty \qquad (-0) * \infty \qquad \infty * NaN$$

(Give signed results where appropriate.)

Exercise 1g

Suppose the IEEE Standard were changed to permit a binary implementation with only 5 bits to represent each number: a sign bit, 2 bits for the exponent, and 2 bits for the precision. Assuming all other features of the Standard are unchanged, including the use of a *hidden bit*, enumerate all 32 possible bit patterns and what they would represent.

2. Elementary numerical methods

2.1 Numerical differentiation

Numerical differentiation is an ill conditioned problem. Nevertheless, it is important because many numerical techniques require approximations to derivatives. It is a good point to start because it introduces in a simple way the ideas of *discretization error*† (error due to approximating a continuous function by a discrete approximation) and *rounding error* (error due to floating-point representation).

2.1.1 Finite differences

The usual definition of the *derivative* $f'(x)$ is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

so an obvious approximation to $f'(x)$ is achieved by choosing a small quantity h and evaluating

$$\tilde{f}'(x) = \frac{f(x+h) - f(x)}{h}. \tag{2.1}$$

We say that $\tilde{f}'(x)$ is a *finite difference* approximation to $f'(x)$. The only problem here is: how small should h be? This turns out to be critical.

Supposing that $f(x)$ can be differentiated at least three times, Taylor's theorem tells us that

$$f(x+h) = f(x) + h.f'(x) + \frac{h^2}{2!}.f''(x) + O(h^3).$$

† Discretization error is often called *truncation error*, but this term is somewhat confusing.

This can be rearranged to give

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}.f''(x) + O(h^2) \tag{2.2}$$

so that, comparing (2.1) and (2.2), the discretization error is approximately $\frac{h}{2}|f''(x)|$ in absolute value.

Writing $[f(x)]^*$ for the floating-point representation of $f(x)$, the calculation involves the evaluation of

$$[f(x+h)]^* = f(x+h) + \varepsilon_{x+h}$$
$$[f(x)]^* = f(x) + \varepsilon_x$$

from which, using (2.1), we get

$$[\tilde{f}'(x)]^* = \tilde{f}'(x) + \frac{\varepsilon_{x+h} - \varepsilon_x}{h}$$
$$= \tilde{f}'(x) + \frac{A}{h}$$

where A is very approximately the absolute error in the representation of $f(x)$, i.e. $macheps.|f(x)|$. The rounding error is therefore approximately $macheps.|f(x)|/h$.

Note that, as h decreases, the discretization error decreases but the rounding error increases. To find the value of h that minimises the total absolute error, we differentiate the right-hand side of the equation

$$total\ absolute\ error = \frac{h}{2}|f''(x)| + \frac{macheps.|f(x)|}{h}$$

with respect to h, and solve

$$\frac{1}{2}|f''(x)| - \frac{macheps.|f(x)|}{h^2} = 0. \tag{2.3}$$

It turns out that this is achieved precisely at the value of h for which $|discretization\ error| = |rounding\ error|$. However, equation (2.3) involves quantities that are not known. For example, we cannot know $f''(x)$ since we are trying to calculate $f'(x)$ in the first place. We are forced to make an approximation, so we assume that $f(x)$ and $f''(x)$ are numbers of order 1 (we sometime write $O(1)$) so we can ignore them in the equation. Then, we might as well also ignore constants, so that equation (2.3) becomes

$$h^2 \simeq macheps$$

or

$$h \simeq \sqrt{macheps}.$$

This is justifiable because it is the order of magnitude of h that is critical, not its exact value. It follows that the total absolute error in the computed derivative is $O(\sqrt{macheps})$. However, as we assumed that $f(x) = O(1)$, it would be more realistic to use $h = \sqrt{macheps}.|f(x)|$ in practice in order that the *relative* error is $O(\sqrt{macheps})$, under appropriate assumptions.

Exercise 2a

List the assumptions made in the analysis in Section 2.1.1. Construct an example for which at least one of these assumptions is unreasonable, and demonstrate why. Why are these assumptions made?

2.1.2 Second derivatives

It is, of course, possible to approximate $f''(x)$ by taking a finite difference of values of $\tilde{f}'(x)$ but, again, care needs to be taken in choice of the value of h.

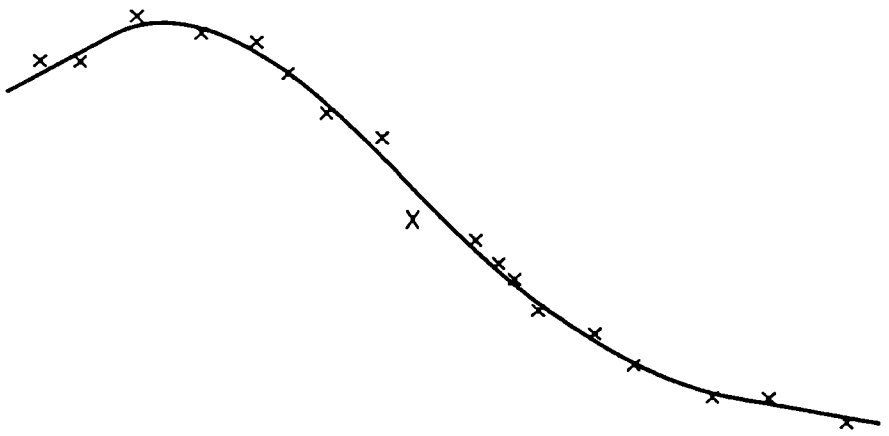
Suppose, for simplicity, that $f(x) = O(1)$ and we use $h = \sqrt{macheps}$ to compute values of $\tilde{f}'(x)$. The relative error in $\tilde{f}'(x)$ is approximately h. As an example, take $macheps = 10^{-16}$. We would use $h = 10^{-8}$ and expect a relative error of 10^{-8} in $\tilde{f}'(x)$.

In calculating $\tilde{f}''(x)$ we must now regard 10^{-8} as if it were the rounding error, so we should use $h = 10^{-4}$ as the optimum value of h, giving an approximate relative error of 10^{-4} in $\tilde{f}''(x)$.

Fortunately, few numerical methods require more than two derivatives. Note that, if $f'(x)$ is known to full floating-point precision, then $\tilde{f}''(x)$ can be calculated with a relative error of about 10^{-8}; this is of importance in designing software interfaces for certain numerical problems.

2.1.3 Differentiation of inexact data

Because numerical differentiation is ill-conditioned, the problem is made much worse if the function $f(x)$ is known only at a finite number of points or if the function values are known only approximately. A little consideration of the above analysis shows that there is no point in attempting to calculate derivatives from the data as it stands. The only sensible course of action is to fit a smooth curve to the data by some means, and to calculate derivatives of the fitted curve, as indicated in the diagram.



It is not possible to make precise statements about the accuracy of the resulting derivatives, but the technique is useful in some circumstances.

A method often used in practice is to perform a *least squares fit* with a *cubic spline*. This is one of many applications of cubic splines, which are described in the next section. Least squares fitting is discussed in Section 2.3.5.

2.2 Splines

We begin with the problem of *interpolation*, that is, to find a curve of a specified form that passes through a given set of data points.

The simplest case is to find a straight line through the pair of points $(x_1, y_1), (x_2, y_2)$. The equation of the

straight line is

$$y = a_1 x + a_0$$

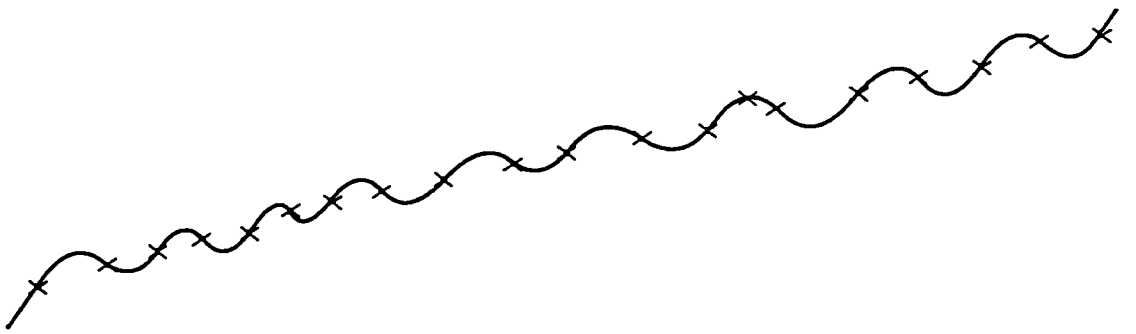
and the problem consists of solving 2 equations (one for each data point) in the 2 unknowns a_0, a_1. We could say that the problem has 2 *degrees of freedom.*

A related problem is to find a straight line that passes through one data point (x_1, y_1), with specified slope m, say. This problem still has 2 degrees of freedom because there are two items, the data point and the slope, that can be varied independently and there are still 2 equations in 2 unknowns.

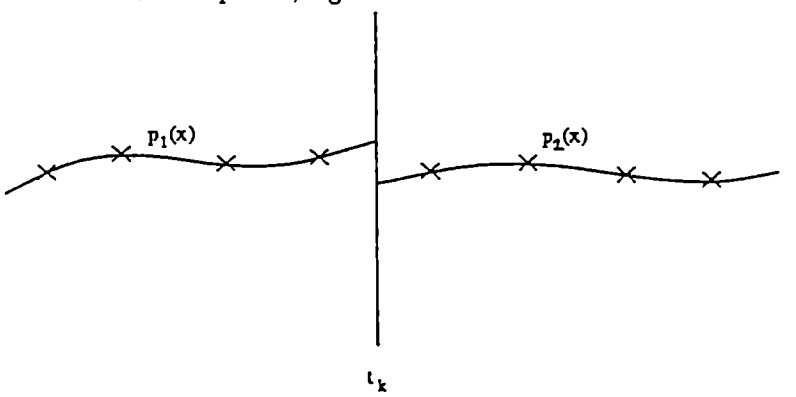
In fitting a quadratic curve there are 3 degrees of freedom. For a cubic curve

$$y = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \tag{2.4}$$

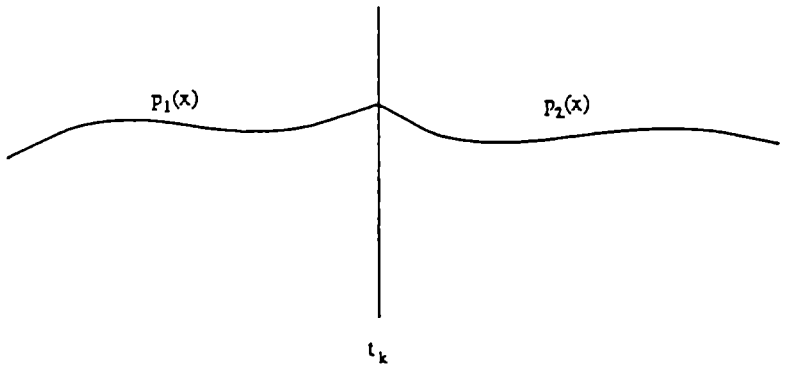
there are 4 degrees of freedom, etc. With 20 points we could, in principle, fit a polynomial of degree 19 but the results might not be as we would wish, e.g.



Low order polynomials do not have this disadvantage. Suppose we decide to interpolate data by fitting two cubic polynomials, $p_1(x)$ and $p_2(x)$, to different parts of the data separated at the point $x = t_k$. We have 8 degrees of freedom and so can fit 8 data points, e.g.



This is worse than using a high order polynomial. What is required is some degree of continuity, e.g. in the curve



if we require that $p_1(t_k) = p_2(t_k)$ then the curve is at least continuous. But this requirement uses up one of our equations, so we now have only 7 degrees of freedom. We can place further constraints as follows:

$$p_1'(t_k) = p_2'(t_k)$$
$$p_1''(t_k) = p_2''(t_k)$$

with an extra degree of freedom taken up by each. This gives a curve with smooth continuity but only 5 degrees of freedom. (If we go a step further and require that $p_1'''(t_k) = p_2'''(t_k)$ then the two cubics become identical and it is equivalent to fitting (2.4).) The point t_k is called a *knot* point. It is possible to specify m such knots and to fit a curve consisting of $m+1$ separate cubics with second derivative continuity. This has $m+4$ degrees of freedom.

A curve made up of cubic polynomials with continuity of the function and first and second derivatives is called a *cubic spline*. Numerical methods often require at least this degree of continuity and a cubic spline is, in some sense, the simplest function that satisfies this requirement.

For the purposes of numerical differentiation of inexact data, fitting a cubic spline and differentiating the result is probably the best that can be achieved, bearing in mind that there is an infinite choice of knot positions.

2.3 Simultaneous linear equations

In this section we consider the solution of simultaneous linear equations of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.5}$$

where $\mathbf{A}$ is a given matrix of coefficients, $\mathbf{b}$ is a given vector, and the vector $\mathbf{x}$ is to be determined. We shall consider here only the case where $\mathbf{A}$ is square and at least one element of $\mathbf{b}$ is non-zero. In this case the equations have a unique solution if and only if $\mathbf{A}$ is a non-singular matrix‡. Mathematically, the solution is given by

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

The first point to note is that there is no need to calculate the inverse $\mathbf{A}^{-1}$ explicitly because the vector $\mathbf{A}^{-1}\mathbf{b}$ can be calculated directly. Also, calculating $\mathbf{A}^{-1}$ and then multiplying by $\mathbf{b}$ leads to unnecessary loss of accuracy. The calculation of a matrix inverse, which is discussed briefly below, is usually avoided unless the elements of the inverse itself are required for some purpose, e.g. in some statistical analyses.

‡ Note that if $\mathbf{A}$ is singular this means that it has no inverse. This is equivalent to saying that $\mathbf{A}$ has zero determinant or that $\mathbf{A}$ has a zero eigenvalue.

The solution of the equations (2.5) is trivial if the matrix $\mathbf{A}$ has either lower triangular form

$$\begin{pmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \cdots & \cdots & \cdots & & \\ a_{n1} & a_{n2} & a_{n3} & & a_{nn} \end{pmatrix}$$

or upper triangular form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ & a_{22} & a_{23} & \cdots & a_{2n} \\ & & a_{33} & \cdots & a_{3n} \\ & & & \cdots & \cdots \\ & & & & a_{nn} \end{pmatrix}$$

where missing coefficients are zero. Note also that if (and only if) any diagonal coefficient a_{ii} is zero then the matrix $\mathbf{A}$ is singular and there is no unique solution.

To see that the solution is trivial to obtain, consider the upper triangular form. In this case the last equation contains only one unknown, x_n, that can therefore be determined by

$$x_n = \frac{b_n}{a_{nn}}.$$

Then, as x_n is now known, the second last equation contains only the one unknown x_{n-1} which may therefore be determined, and so on.

This so-called *back substitution* algorithm may be summarised as

$$x_i := \frac{b_i - \sum_{j=i+1}^{n} a_{ij} x_j}{a_{ii}}$$

for $i = n, n-1, \ldots 1$. (Note that, by convention, the summation is over zero terms when the limits do not make sense.)

For a general set of equations $\mathbf{Ax} = \mathbf{b}$ the solution $\mathbf{x}$ is unchanged if any of the following operations is performed:

(1) Multiplication of an equation by a non-zero constant.

(2) Addition of one equation to another.

(3) Interchange of two equations.

The technique used for solving a general set of equations $\mathbf{Ax} = \mathbf{b}$ is called *Gaussian elimination* and consists of using a combination of the operations (1), (2) and (3) to convert the equations to a trivial case, by convention the upper triangular form. There are a large number of different ways of achieving this. The strategy adopted is usually called the *pivotal strategy*, and we shall see that this is crucially important.

We consider two examples.

Example 5

This example is a well-conditioned problem that illustrates the need for a pivotal strategy. Consider the 3 simultaneous equations

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix}$$

which are to be converted to upper triangular form. Note that the first equation cannot form the first row of the upper triangle because its first coefficient is zero. Therefore we must first interchange the first two equations, thus

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}.$$

The next step is to subtract the (new) first equation from the third to get

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}.$$

The final step is to subtract the second equation from the third and the solution $x_1 = \frac{5}{2}$, $x_2 = \frac{3}{2}$, $x_3 = -\frac{1}{2}$ results.

This is a fairly trivial exercise on a sheet of paper, but it indicates that a computer subroutine to deal with the general case is less straightforward because several coefficients may be zero at any step. A successful pivotal strategy must specify which of the operations (1), (2) and (3) to perform on which equations at each stage of the calculation.

Example 6

This example shows that it is not only zero coefficients that cause trouble. Consider the pair of equations

$$\begin{pmatrix} 0.0003 & 1.566 \\ 0.3454 & -2.436 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1.569 \\ 1.018 \end{pmatrix}$$

which have the *exact* solution $x_1 = 10$, $x_2 = 1$. However, if we perform the calculation on a computer with *macheps* $\simeq 0.5 \times 10^{-4}$, and multiply the first equation by $0.3454/0.0003$ ($\simeq 1151$) and subtract from the second, we get the solution $x_1 = 3.333, x_2 = 1.001$ because of loss of significance. The use of a small non-zero number as a *pivotal value* has led to an incorrect answer.

In fact, closer examination of the loss of significance shows that it is due to the fact that the coefficient 0.0003 is small *compared with* 1.566.

Therefore a successful pivotal strategy requires some concept of the *size* of a coefficient *relative* to the other coefficients in the same equation. This relative size is conveniently calculated by dividing each coefficient by the l_∞ norm of the corresponding row of the matrix. In other words, we *scale* each equation by dividing it by its largest coefficient (in absolute value). In Example 6, we would divide the first equation by 1.566 and the second by 2.436. This provides a simple automatic way of checking whether a coefficient is (relatively) small.

2.3.1 Pivotal strategies

Suppose an $n \times n$ set of linear equations is being solved by Gaussian elimination. At the 1st step there are n possible equations and one is chosen as pivot. This eliminates one variable and leaves $n-1$ equations at the 2nd step. At the start of the kth step there are therefore $n-k+1$ remaining equations from which to select a pivot.

In *partial pivoting* these $n-k+1$ equations are scaled, by dividing by the largest coefficient of each, then the pivotal equation is chosen as the one with the largest (scaled) coefficient of x_k.

In *total pivoting* both the pivotal equation and pivotal variable are selected by choosing the largest (unscaled) coefficient of any of the remaining variables. Total pivoting may be better than partial pivoting, although it is more expensive and so is not often used.

Work in the 1940s, using forward error analysis, had suggested that the error in Gaussian elimination grew at a rate that would make the process unstable for large systems of equations. In the 1950s J.H. Wilkinson used backward error analysis to show that this predicted rate of error growth was merely the worst case, and not one that appears to occur in practical problems. Gaussian elimination is generally *stable* for *well-conditioned* problems.

Exercise 2b

List the basic operations of Gaussian elimination, i.e. the three operations on the equations $\mathbf{Ax} = \mathbf{b}$ that leave the solution $\mathbf{x}$ unchanged. Illustrate how these operations are combined, using exact arithmetic on the matrix

$$\begin{pmatrix} 0 & 1 & 3 \\ 2 & -2 & 1 \\ 4 & 1 & 0 \end{pmatrix}$$

Show how the method is modified for floating-point arithmetic, by using *partial pivoting* on the matrix

$$\begin{pmatrix} 4 & 1 & 2 \\ 0 & 10^{-4} & 3 \\ 1 & 10.25 & 4 \end{pmatrix}$$

(In each case, omit the back substitution phase of the calculation.)

2.3.2 Symmetric positive definite equations

Many practical numerical problems reduce ultimately to the solution of sets of simultaneous linear equations. For difficult problems, e.g. the solution of a partial differential equation in a complicated 2- or 3-dimensional region, the resulting linear equations may be huge, say 10000 simultaneous equations.

In general, such large systems may be arbitrarily ill-conditioned and the resulting matrix may *appear to be singular* when floating point arithmetic is used. Worse still, the relative errors may become much greater than 1 so that incorrect solutions will be obtained. Indeed even a 2×2 system can lead to inaccurate answers if the matrix is 'nearly' singular.

However, an important class of problems is always well-conditioned, and Gaussian elimination gives reliable answers for any size of matrix. These are problems in which the matrix is *symmetric positive definite*, and pivoting is not needed.

A symmetric matrix $\mathbf{A}$ is *positive definite* if it satisfies the inequality

$$\mathbf{x}^T\mathbf{Ax} > 0$$

for **any** non-zero vector $\mathbf{x}$. For example, if

$$\mathbf{A} = \begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix}$$

then

$$\begin{aligned} \mathbf{x}^T\mathbf{Ax} &= (x_1 \quad x_2)\begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ &= 4x_1^2 + 2x_1x_2 + 4x_2^2 \\ &= (x_1 + x_2)^2 + 3(x_1^2 + x_2^2) \\ &> 0 \end{aligned}$$

in all cases. It is not always easy to verify this property, but an important special case is where $\mathbf{B}$ is any real non-singular matrix and $\mathbf{A} = \mathbf{B}^T\mathbf{B}$, then

$$\mathbf{x}^T\mathbf{Ax} = \mathbf{x}^T\mathbf{B}^T\mathbf{Bx} = (\mathbf{Bx})^T\mathbf{Bx} > 0.$$

Fortunately, good algorithms for symmetric positive definite equations can detect when a matrix is not, in fact, positive definite so it is not essential to check this independently before attempting a calculation.

Many important practical problems give rise to symmetric positive definite equations.

2.3.3 Choleski factorization

An instructive way of looking at Gaussian elimination is to consider it in terms of matrix algebra. This leads to a useful algorithm for the solution of symmetric positive definite equations.

When solving $\mathbf{Ax} = \mathbf{b}$ by Gaussian elimination the matrix $\mathbf{A}$ is transformed to an upper triangular matrix $\mathbf{U}$ by means of operations which include the permutation of rows and columns of the matrix.

A *permutation matrix*, call it $\mathbf{P}$, is simply a matrix that permutes another matrix when it is multiplied by $\mathbf{P}$; its elements are all either 0 or 1 in such a way that there is exactly one 1 in each row and column. The unit matrix $\mathbf{I}$ is thus the (trivial) permutation matrix that does not change anything.

It may be shown that Gaussian elimination is equivalent to factorizing the matrix $\mathbf{A}$ such that

$$\mathbf{A} = \mathbf{PLU}$$

where $\mathbf{L}$ is a lower triangular matrix. The equations $\mathbf{Ax} = \mathbf{b}$ may then be written $\mathbf{PLUx} = \mathbf{b}$ which means that the resulting upper triangular system may be expressed as

$$\mathbf{Ux} = \mathbf{L}^{-1}\mathbf{P}^{-1}\mathbf{b}$$

where the matrices $\mathbf{P}$ and $\mathbf{L}$ are straightforward to invert.

Choleski factorization may be applied to symmetric positive definite matrices and differs from the $\mathbf{PLU}$ factorization in three ways:

(1) Because symmetric positive definite equations are well conditioned no permutations are necessary so $\mathbf{P} = \mathbf{I}$, i.e. $\mathbf{P}$ can be omitted.

(2) By symmetry it is possible to arrange that $\mathbf{U} = \mathbf{L}^T$ so the factorization can be expressed in the form $\mathbf{A} = \mathbf{LL}^T$. However, the diagonal elements of $\mathbf{L}$ involve taking square roots; this can be avoided.

(3) In order to avoid taking square roots the diagonal elements of $\mathbf{L}$ are defined to be 1 (and so do not need to be stored) and the factorization used is

$$\mathbf{A} = \mathbf{LDL}^T$$

where $\mathbf{D}$ is a diagonal matrix. This is the Choleski factorization.

The algorithm for solving symmetric positive definite equations is then as follows: form the Choleski factorization, then solve

$$\mathbf{Ly} = \mathbf{b}$$
$$\mathbf{L}^T\mathbf{x} = \mathbf{D}^{-1}\mathbf{y}$$

2.3.4 Matrix inversion

Suppose that the two sets of equations $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Ay} = \mathbf{c}$ are to be solved, i.e. two sets of equations with the same coefficient matrix. This problem could be written

$$\mathbf{A}(\mathbf{x} \quad \mathbf{y}) = (\mathbf{b} \quad \mathbf{c})$$

which can be solved in one go by Gaussian elimination. Note that both the solution and the right-hand side are now matrices rather than vectors. In the same way, one application of Gaussian elimination is sufficient

to solve the matrix equation $\mathbf{AX} = \mathbf{C}$ where $\mathbf{X}$ is an unknown matrix and $\mathbf{C}$ is a constant matrix of the same shape as $\mathbf{X}$.

In particular, if we take $\mathbf{C}$ to be the unit matrix $\mathbf{I}$ then the solution $\mathbf{X}$ will be $\mathbf{A}^{-1}$.

As stated above, it is not usually necessary or desirable to invert a matrix, but sometimes the elements of the inverse itself are required and Gaussian elimination is normally used as described.

2.3.5 Linear least squares

Problems are often encountered where it is desirable to find an n-dimensional vector, call it $\mathbf{x}_n$, that most closely satisfies (in some sense) the set of m equations

$$\mathbf{A}_{mn}\mathbf{x}_n = \mathbf{b}_m \tag{2.6}$$

where $\mathbf{A}_{mn}$ is an $m \times n$ matrix. This is an *overdetermined* system and is usually solved in the *least squares* sense. Define the residual vector

$$\mathbf{r}_m = \mathbf{A}_{mn}\mathbf{x}_n - \mathbf{b}_m.$$

Then the least squares solution of (2.6) is the vector $\mathbf{x}_n$ that minimises the sum of squares of the residuals, i.e.

$$\mathbf{r}^T\mathbf{r} = (\mathbf{Ax} - \mathbf{b})^T(\mathbf{Ax} - \mathbf{b}).$$

The least squares solution may be shown to be the solution of the equations

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$$

where $\mathbf{A}^T\mathbf{A}$ is square and symmetric positive definite. These so-called *normal equations* may be solved by Choleski factorization. However, since the mid 1970s, this approach has been replaced by solving directly the overdetermined system $\mathbf{Ax} = \mathbf{b}$ by a method known as *singular value decomposition*†.

Singular value decomposition is a stable method that can be used to solve almost any system of linear equations, yielding a unique solution. In the case that the matrix is singular, the solution will be in a lower dimensional space than the original problem. It should be emphasised that such solutions may or may not have physical meaning in a given context. The technique is most useful for non-square matrices, and even underdetermined problems.

2.4 Non-linear equations

Having dealt with linear equations, we will now look briefly at the solution of simultaneous non-linear equations. An example of a system of 2 equations with 2 unknowns is

$$\begin{aligned} x_1 + \sin x_2 - \frac{8}{7} &= 0 \\ x_1 x_2 - \frac{15\pi}{28} &= 0 \end{aligned} \tag{2.7}$$

which has a solution $x_1 = 9/14$, $x_2 = 5\pi/6$, which is easily verified. However there are two other solutions with x_2 values close to $5\pi/2$. This is an extra complication: non-linear equations do not, in general, have a unique solution. In fact, a set of non-linear equations can have any number of solutions; there may be no solutions at all, or there may be infinitely many.

---

† The details of this method are beyond the scope of this course.

Exercise 2c

Show, graphically or otherwise, that there are exactly three pairs of solutions to the equations (2.7).

It is generally true in numerical analysis that problems without a unique solution are more difficult. Such problems must usually be solved iteratively, by using a starting approximation close (in some sense) to the required solution. Two particular difficulties that arise are (a) convergence to the 'wrong' solution, and (b) failure to converge, sometimes because approximations oscillate between two or more solutions.

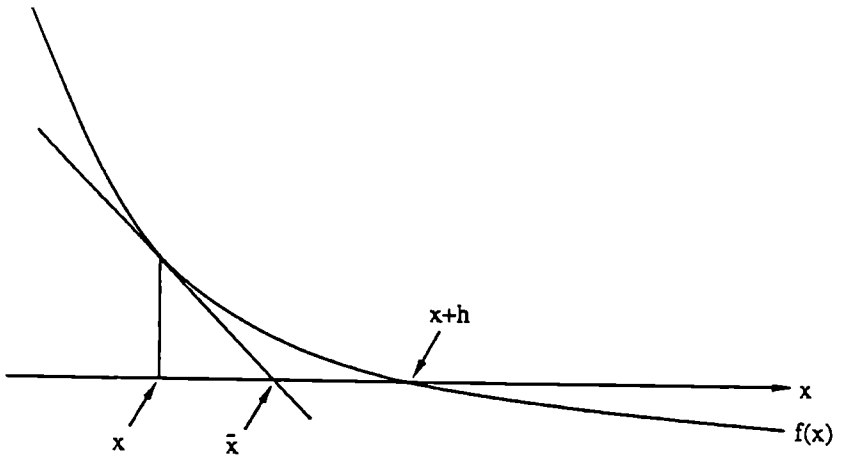
A set of n non-linear equations in n unknowns $x_1, x_2, \ldots x_n$ may be written

$$\begin{aligned} f_1(x_1, x_2, \ldots x_n) &= 0 \\ f_2(x_1, x_2, \ldots x_n) &= 0 \\ &\cdots \\ f_n(x_1, x_2, \ldots x_n) &= 0 \end{aligned}$$

which can be abbreviated to $\mathbf{f}(\mathbf{x}) = 0$ where $\mathbf{x}$ is the vector of unknowns and $\mathbf{f}$ is the vector of functions f_j.

2.4.1 Newton-Raphson iteration

We consider first the equation $f(x) = 0$ where $f(x)$ is a smooth non-linear function of a scalar x. The *Newton-Raphson method* is an iterative technique that is graphically equivalent to improving the estimated solution by moving along the gradient of the function $f(x)$, evaluated at the point x.



The formula may be derived from the Taylor series for $f(x)$ from which we get

$$f(x + h) = f(x) + hf'(x) + O(h^2). \tag{2.8}$$

If x is an approximate solution and $x + h$ is the exact solution then $f(x + h) = 0$ and we can estimate the value of h from (2.8) to get

$$h = -\frac{f(x)}{f'(x)} + O(h^2).$$

Writing $\bar{x}$ to represent an improved approximation we have

$$\bar{x} = x - \frac{f(x)}{f'(x)}. \tag{2.9}$$

The error in this approximation is $O(h^2)$ so it has a quadratic rate of convergence. In fact this rate of convergence is only obtained when the approximation is sufficiently close to the solution. Note that (2.9) could be written in either of the alternative forms

$$\tilde{x} = x - [f'(x)]^{-1} f(x)$$

$$f'(x)(\tilde{x} - x) = -f(x).$$

As a simple example of the use of (2.9), consider the calculation of $\sqrt{2}$. This is equivalent to solving the equation $f(x) = 0$ where $f(x) = x^2 - 2$. In this case $f'(x) = 2x$ and (2.9) becomes

$$\begin{aligned}\tilde{x} &= x - \frac{x^2 - 2}{2x} \\ &= \frac{x^2 + 2}{2x} \\ &= \frac{x}{2} + \frac{1}{x}.\end{aligned}$$

Using $x = 1$ as a starting approximation, successive values of $\tilde{x}$ (obtained on a BBC micro) are 1.5, 1.41666667, 1.41421569, 1.41421356. (Squaring the last approximation gave 2 to the accuracy of the floating point arithmetic.)

Exercise 2d

The Newton-Raphson formula (2.9) was compared with

$$\hat{x} = x - \frac{hf(x)}{f(x+h) - f(x)} \tag{2.10}$$

for solution of the equation $x^2 - 5 = 0$. Using a particular value of h, the following numerical results were obtained for the first three iterations of each method using the starting value $x = 2$ in each case:

iteration	method(2.9)	method(2.10)
0	2.000000000	2.000000000
1	2.250000000	2.235988201
2	2.236111111	2.236063956
3	2.236067978	2.236067775

For the first two iterations method (2.10) was more accurate, but method (2.9) was more accurate on the third iteration. Which method would you expect to converge faster, and why?

Suppose method (2.10) is used on a computer for which *macheps* is about 10^{-24}. Discuss a suitable choice of the parameter h.

2.4.2 Generalisation of Newton-Raphson

Newton-Raphson iteration can be generalised for solving n non-linear equations in n unknowns, i.e. for solving the vector equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. The derivative $f'(x)$ generalises to a matrix of partial derivatives, the (i,j)th element of which is

$$\frac{\partial f_i(\mathbf{x})}{\partial x_j}$$

for $i = 1, 2, \ldots n$ and $j = 1, 2, \ldots n$. This matrix of derivatives in called the *Jacobian* $\mathbf{J}$. As division is not defined for matrices, we must use one of the alternative forms of (2.9) to express the Newton-Raphson formula, i.e.

$$\tilde{\mathbf{x}} = \mathbf{x} - \mathbf{J}^{-1}\mathbf{f}$$

or, in order to express this as the solution of a set of *linear* equations,

$$\mathbf{J}(\tilde{\mathbf{x}} - \mathbf{x}) = -\mathbf{f}. \tag{2.11}$$

Note that the solution $\tilde{\mathbf{x}} - \mathbf{x}$ of (2.11) is the vector of increments to be added to $\mathbf{x}$ to form the new approximation.

The rate of convergence of the generalised Newton iteration can also be shown to be quadratic if the approximation is sufficiently accurate but, unfortunately, multi-dimensional problems are more difficult and it is much harder to ensure convergence. Algorithms for non-linear equations usually involve Newton-Raphson iteration, often as a final step, but more strategy and care is needed than in the scalar problem.

Nevertheless, to illustrate that the formula (2.11) is of some use, we solve the pair of equations (2.7). Here the Jacobian is

$$\begin{pmatrix} 1 & \cos x_2 \\ x_2 & x_1 \end{pmatrix}.$$

The starting approximations $x_1 = 1$, $x_2 = 3$ yield successive approximations

$$\begin{aligned} x_1 &= 0.672016558 \quad x_2 = 2.66694639 \\ x_1 &= 0.643149516 \quad x_2 = 2.61895767 \\ x_1 &= 0.642857175 \quad x_2 = 2.61799418 \\ x_1 &= 0.642857143 \quad x_2 = 2.61799388 \end{aligned}$$

where the last pair of values is equal to $9/14$, $5\pi/6$ to the accuracy of the arithmetic. The starting values $x_1 = 1$, $x_2 = 7$ lead to convergence to $x_1 = 0.226117142$, $x_2 = 7.4430273$, and starting values $x_1 = 1$, $x_2 = 8$ lead to convergence to $x_1 = 0.205031727$, $x_2 = 8.20846651$ corresponding to the other two solutions. However, it is all too easy to find starting values which do not lead to convergence. For example, taking a starting value in between the latter two solutions yields particularly misleading results. If $x_1 = 0.2$, $x_2 = 5\pi/2$ then successive convergents 'settle down' in the region of $x_1 = 0.008$, $x_2 = 20$ although only the crudest of error tests would be deceived into falsely detecting convergence. There is no solution of the equations near these values, although a little graphical consideration will show why the algorithm tries to find one.

Exercise 2e

> Investigate this problem, both graphically and by writing a short program, using any language with a floating-point data type, to find starting values that lead to convergence and otherwise.

It should be noted that it is often cheaper (in terms of programmer time or computation time) to use numerical derivatives to estimate the Jacobian matrix $\mathbf{J}$ in (2.11) but the resulting loss of accuracy may itself cause problems.

2.4.3 Ill-conditioned problems

We now return to the scalar non-linear equation $f(x) = 0$ to investigate cases in which the problem is ill-conditioned; these are often cases in which Newton-Raphson iteration becomes unstable. One such case is where $f'(x) = 0$ at the solution of $f(x) = 0$ such that a Newton-Raphson step performed at the solution

would result in division by zero. A typical example is shown below:

Note that if the x-axis were shifted downwards there would be no solution, and if shifted upwards there would be two solutions. An ingenious way of overcoming the difficulty is to solve a sequence of (different) problems which successively get closer to the original problem. This sequence of problems is chosen such that each is better conditioned than the original problem. Techniques of the type are known as *continuation methods.*

In order to solve a difficult case of the problem $f(x) = 0$, with a starting approximation x_0, we first solve

$$f(x) = \lambda f(x_0) \tag{2.12}$$

for some parameter $0 < \lambda < 1$. Note that the original problem corresponds to $\lambda = 0$ and that the problem $\lambda = 1$ is guaranteed to have a solution ($x = x_0$). By taking a succession of decreasing values of λ and using a suitable *robust* algorithm (not Newton-Raphson) the solution is obtained to a sequence of problems that get nearer to the original problem. Usually, as λ approaches zero, these solutions tend to the required solution.

Note that an algorithm may be described as *robust* if it can be guaranteed to be successful for any valid data. This is often achieved at the expense of time.

2.5 Quadrature

2.5.1 Quadrature rules

Quadrature is the name given to the numerical evaluation of an integral of the form

$$I = \int_a^b f(x)dx$$

where either (or both) of a or b may possibly be infinite. An approximation of the form

$$\int_a^b f(x)dx \simeq \sum_{i=0}^{n} w_i f(x_i)$$

where the points x_i are chosen such that $a \leq x_0 < x_1 < \ldots, x_n \leq b$, is called a *quadrature rule.* The points x_i are called the *abscissae* and the numbers w_i are called *weights.*

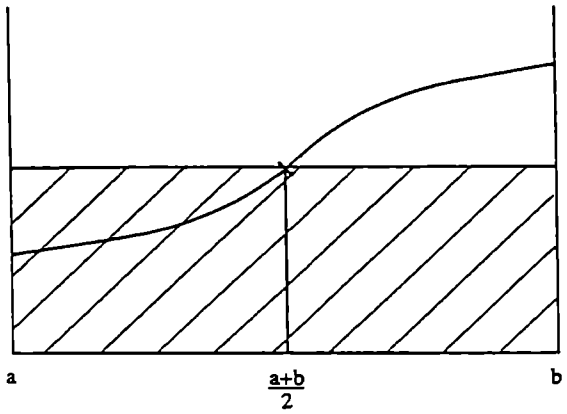
Quadrature rules are usually derived by the integration of an interpolating polynomial. Normally, to avoid loss of significance, only rules with positive weights are used.

We define the error in a quadrature rule by

$$e_n = I - \sum_{i=0}^{n} w_i f(x_i)$$

and it is possible to obtain an expression for this from the corresponding interpolating polynomial ‡.

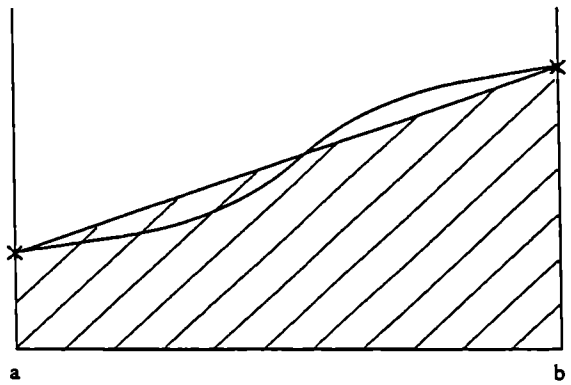
We will now examine a few simple quadrature rules to see how their error terms compare. We consider the case where a and b are both finite. The simplest reasonable quadrature rule is obtained by taking the value of $f(x)$ at the mid-point of the interval (a, b) and treating the function as if it were a constant:



Not surprisingly, this is called the *mid-point rule*. The formula, including the error term, is

$$I = (b-a)f(\frac{a+b}{2}) + \frac{f''(\xi)(b-a)^3}{24} \tag{2.13}$$

for some ξ where $a < \xi < b$. Alternatively we can fit a straight line between the function values at the end-points of the interval:



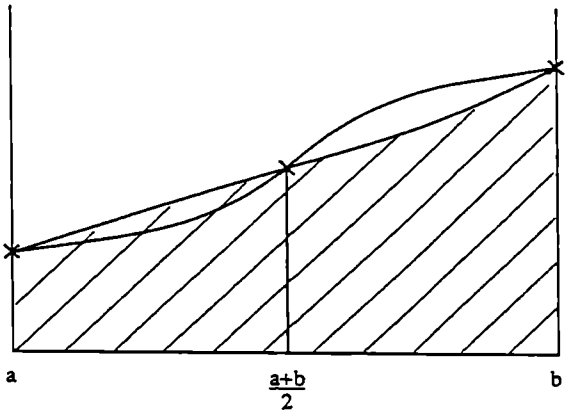
This is the trapezium rule with the formula

$$I = \frac{b-a}{2}[f(a) + f(b)] - \frac{f''(\eta)(b-a)^3}{12} \tag{2.14}$$

for some η where $a < \eta < b$. A spectacular improvement is obtained by a fitting a quadratic curve through

‡ See Conte & de Boor.

the mid-point and the end-points:



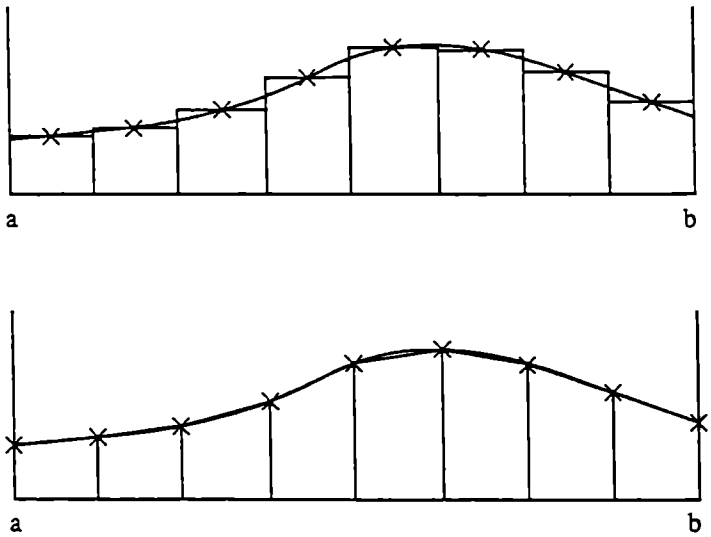
This leads to *Simpson's rule* which is

$$I = \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)] - \frac{f^{iv}(\zeta)(\frac{b-a}{2})^5}{90} \tag{2.15}$$

for some ζ where $a < \zeta < b$.

This idea can be extended by, say, interpolating four equally-spaced points with a cubic polynomial. Quadrature rules derived from equally-spaced points are called *Newton-Cotes formulae*. High order Newton-Cotes rules are rarely used for two reasons: (i) in high order rules some weights are negative, which leads to instability, and (ii) methods based on high order polynomials usually have the same disadvantages as high order polynomial interpolation.

A more promising approach than increasing the order of the interpolating polynomial is to use a *composite rule*. A *composite rule* is formed by splitting an integral into a set of panels and applying (usually) the same quadrature rule in each panel and summing the results. For example the composite mid-point and composite trapezium rules:



Let there be n panels. Writing h for the width of each panel, these lead to the similar formulae

$$I \simeq h \sum_{i=1}^{n} f(a + \{i - \frac{1}{2}\}h)$$

for the composite mid-point rule, and

$$I \simeq h \sum_{i=0}^{n}{}'' f(a+ih)$$

for the composite trapezium rule, where the double prime on the summation indicates that the first and last terms are each halved. The fact that these formulae are similar helps to explain why the error terms for these rules are also similar.

Replacing $b-a$ by h in (2.13) and (2.14) we see that these rules each have order of convergence h^3 in any one panel. But there are n panels, and $n = O(h^{-1})$, so the order of convergence of the composite rule is $h^{-1}.h^3 = h^2$.

The composite Simpson rule has the formula

$$I \simeq \frac{h}{6}[(f(a)+f(b)+2\sum_{i=1}^{n-1} f(a+ih)+4\sum_{i=1}^{n} f(a+\{i-\frac{1}{2}\}h)]$$

which, from (2.15), is of order $h^{-1}.h^5 = h^4$.

In general, a single quadrature rule derived from an interpolation formula based on n abscissae is guaranteed to have order at least h^{n+1}, although formulae based on specially advantageous choices of the abscissae have higher orders of convergence. Note that both the mid-point rule and Simpson's rule have one higher order of convergence than expected; the trapezium rule has the minimum order. The maximum possible order of convergence for a polynomial rule based on n abscissae is h^{2n+1}; such a rule is called a *Gauss rule*† and uses optimally chosen abscissae. These abscissae turn out to be non-equally spaced points which, incidentally, never include the end-points of the interval; for odd values of n the mid-point is always one of the Gauss abscissae.

Because an arbitrary function $f(x)$ is potentially expensive to calculate, the efficiency of a quadrature scheme is usually measured by the number of function evaluations required to estimate the integral to a specified accuracy. In a composite rule it is therefore an advantage for the end-points to be abscissae, because a function evaluation at an end-point of one panel is used again in the next panel. This also explains why there is so little difference between the composite mid-point and composite trapezium rules in practice.

An interesting compromise between efficiency and high order is achieved by a *Lobatto rule*. This is an n-point quadrature rule in which the end-points are included and the remaining $n-2$ abscissae are chosen to achieve the maximum possible order of convergence (which then turns out to be h^{2n-1}). The importance of Simpson's rule is explained by the fact that it is the simplest Lobatto rule.

2.5.2 Summation of series

This section gives a simple example of the use of numerical analysis to devise and tune an algorithm for solving a problem that is easy to pose, but difficult to solve by straightforward means. The method uses the mid-point formula (2.13) in order to sum a series by evaluating an integral: this may be thought of as quadrature in reverse‡. In order to simplify (2.13) a little, first consider the integral of some function $f(x)$ over an interval of unit length. Writing

$$I_n = \int_{n-\frac{1}{2}}^{n+\frac{1}{2}} f(x)dx$$

we see that (2.13) simplifies to

$$I_n = f(n) + \frac{f''(\theta_n)}{24}$$

† The derivation of Gauss rules is discussed in Conte & de Boor.

‡ The method described is also related to the so-called *Euler-Maclaurin summation formulae.*

where θ_n is some (unknown) value in the interval $(n-\frac{1}{2}, n+\frac{1}{2})$. Let e_n be the error term, such that

$$e_n = I_n - f(n) = \frac{f''(\theta_n)}{24}.$$

In the case where $f''(x)$ happens to be a positive decreasing function in the interval $(n-\frac{1}{2}, n+\frac{1}{2})$, we can certainly say that

$$|e_n| \leq \frac{f''(n-\frac{1}{2})}{24}.$$

Using the notation

$$S_{p,q} = \sum_{n=p}^{q} f(n), \tag{2.16}$$

we now consider the problem of evaluating the infinite series $S_{1,\infty}$ where, for sufficiently large values of x, $f(x)$ is a positive decreasing function of x and a formula can be found for $\int f(x)dx$. Of particular interest are functions for which $S_{1,\infty}$ is not easy to evaluate by simple summation alone.

We start by devising an algorithm applicable to all suitable functions $f(x)$, then use a particular example to see how the algorithm may be tuned for efficiency. Note that a sum of the form (2.16) is equivalent to application of the composite mid-point rule to an integral. Suppose that we start off the calculation of $S_{1,\infty}$ by summing N terms then use the composite mid-point rule to approximate the remainder of the sum. We have

$$\begin{aligned} S_{1,\infty} &= S_{1,N} + S_{N+1,\infty} \\ &= S_{1,N} + \sum_{n=N+1}^{\infty} (I_n - e_n) \\ &= S_{1,N} + \int_{N+\frac{1}{2}}^{\infty} f(x)dx - \frac{1}{24}\sum_{n=N+1}^{\infty} f''(\theta_n) \\ S_{1,\infty} &= S_{1,N} + \int_{N+\frac{1}{2}}^{\infty} f(x)dx + E_N \end{aligned} \tag{2.17}$$

where, if $f''(x)$ is a positive decreasing function for $x > N+\frac{1}{2}$, the error estimate is such that

$$\begin{aligned} |E_N| &\leq \frac{1}{24}\sum_{n=N+1}^{\infty} f''(n-\frac{1}{2}) \\ &\simeq \frac{1}{24}\int_{N}^{\infty} f''(x)dx \\ |E_N| &\simeq -\frac{f'(N)}{24}. \end{aligned} \tag{2.18}$$

Note that, given a target absolute error ε, if the integral in (2.17) can be evaluated then it is possible to determine a value of N that should achieve this prescribed error given sufficient floating-point precision.

As a specific example of a well-known function that is hard to evaluate directly, consider the *Riemann zeta function*:

$$\zeta(k) = \sum_{n=1}^{\infty} n^{-k}$$

for any $k > 1$. Writing $f(n) = n^{-k}$, the integral in (2.17) is

$$\int_{N+\frac{1}{2}}^{\infty} x^{-k}dx = \left[\frac{x^{1-k}}{1-k}\right]_{N+\frac{1}{2}}^{\infty} = \frac{(N+\frac{1}{2})^{-(k-1)}}{k-1}$$

and note that $f''(x)$ is positive and decreases with x. From (2.18) we get that $|E_N| \simeq \frac{k}{24} N^{-(k+1)}$ and we estimate the value of N necessary to achieve the target absolute error ε as

$$N = \left\lfloor \frac{k}{24\varepsilon} \right\rfloor^{\frac{1}{k+1}}. \tag{2.19}$$

The formula to be used is then

$$\zeta(k) = \sum_{n=1}^{N} n^{-k} + \frac{(N+\frac{1}{2})^{-(k-1)}}{k-1} + E_N \tag{2.20}$$

where $|E_N| \simeq \varepsilon$.

As a numerical example, suppose $k = 2$ and the target absolute error $\varepsilon = 10^{-16}$. With straightforward summation, we can use the integral $\int_{N+\frac{1}{2}}^{\infty} f(x)dx$ to estimate how many terms are needed: this works out at about 10^{16}. Using (2.19) and (2.20) we can estimate the value

$$N = \left\lfloor \frac{2}{24.10^{-16}} \right\rfloor^{\frac{1}{3}} \simeq 10^5$$

for our algorithm. When $k = 2$, if a computer takes, say, about 0.1 second for this calculation then the straightforward sum would take over 300 years.

Finally, it is worth noting that the integral in (2.17) can itself be estimated by quadrature, which allows the method to be used even when a formula for $\int f(x)dx$ is not available. However, the quadrature rule will add to the computational cost of the algorithm and will complicate the error formula.

Exercise 2f

Implement the algorithm described in Section 2.5.2, in any language with a floating-point data type. Use the following known values to test its accuracy:

$$\zeta(2) = \frac{\pi^2}{6}$$
$$\zeta(4) = \frac{\pi^4}{90}$$

(If your floating-point implementation has poor numerical properties then you may need to compute the sum carefully to achieve the expected accuracy – recall Exercise 1a.)

If a program timer is available compute $\zeta(2)$, say 10 times, and hence estimate how long it would take to compute the value once, to similar accuracy, using a straightforward sum.

3. Numerical software

3.1 State of the art

For mainly historical reasons, most software for numerical methods is found in subroutine libraries designed to be used from conventional programming languages. This is in contrast to, say, statistical software which is often provided in the form of a free-standing package. There are a number of special-purpose libraries for single application areas, such as linear equations or optimization. Some research organisations have produced general-purpose libraries but these are often restricted to a particular range of computers, or have gaps in areas that are not of interest at the originating site. There are only two portable general purpose libraries currently in wide use on a world scale. These are the libraries of NAG (Numerical Algorithms Group), based in Oxford, and Precision Visuals (formerly IMSL), based in the United States.

3.2 Languages

It is not possible to proceed very far with this subject before discussing the vexed question of the languages in which numerical software has been implemented. The significant languages used up to now have been: Algol (both Algol 60 and Algol 68), Fortran (all dialects), Pascal and Ada. In Autumn 1990 NAG released a small library written in C. By far the most commonly used language for numerical software is Fortran, for largely historical reasons. Whereas Fortran has few redeeming features as a programming language, it does have standardisation and portability in its favour. Also, in view of the substantial body of well-tested software that already exists in Fortran, it can be expected to maintain its pre-eminent position for a few years at least. A new Standard, Fortran 90, was standardised in 1991 and NAG has produced the first compiler, though not as yet accompanied by a numerical library.

In the 1980s, it was predicted that Algol 68, Pascal or Ada would become the future language for numerical software, but none of these forecasts has come true. Algol 68 has virtually died out, and never was much used in the United States. The difficulty with standard Pascal is that 2-dimensional arrays, crucial to the handling of matrices in numerical subroutines, are not implemented in a suitable way. There was heavy investment in Ada by the N.A.T.O. countries for military purposes, but the end of the 'Cold War' has resulted in a rapid decline in interest in this language for largely non-technical reasons.

It is to be hoped that good numerical subroutines will become widely available from good programming languages in the future, although this is unfortunately not the case at present. The future trend may be to improve facilities for mixed language programming, or possibly to move towards packaged numerical software.

3.3 Floating-point arithmetic: software considerations

3.3.1 The Brown model

We have already seen that one implementation-specific constant, *macheps*, is important in numerical algorithms. In order to circumvent overflow or underflow in an algorithm it is also necessary to know the limits of the representable number range. Other constants may be significant in particular algorithms.

It is still the case that IEEE implementations are the exception rather than the rule. The task of writing implementation-independent subroutines is considerably simplified by the use of a 'lowest common denominator' model of floating-point arithmetic. The Brown model‡, or some modification of it, is one that is commonly used. The model assumes that a floating-point representation can be described by only four parameters:

B - the base of the arithmetic

N - the number of digits (of base B) in the mantissa

$EMIN$ - the minimum exponent

$EMAX$ - the maximum exponent

then any floating-point number can be represented in the form

$$\pm(.d_1 d_2 \ldots d_N)B^E$$

where $d_1 d_2 \ldots d_N$ are digits (of base B) and E is the exponent in the range $EMIN \leq E \leq EMAX$. A library need only contain four implementation-dependent functions that supply the values of these parameters for the use of portable algorithms. In principle, other implementation-dependent constants can be derived from the parameters of the Brown model.

‡ Due to W.S. Brown, 1981.

3.3.2 Implementation issues for IEEE arithmetic

IEEE arithmetic is more closely specified than Brown's model, which was used to define floating-point arithmetic in the language Ada. For example under IEEE it is possible to prove that $(3.0/10.0) * 10.0$ evaluates to 3.0; this is not possible under the axioms of Brown's model.

Under IEEE 754, because it specifies the number of bits to be used, the use of exactly rounded operations means that the results of calculations should be identical when a program is moved from one processor to another, provided they both support the Standard. Furthermore it is possible, at least in principle, to prove that some floating-point algorithms produce answers correct to within prescribed tolerances†.

Another difficulty that is encountered in programming languages, e.g. some of the early implementations of C (i.e. Kernighan & Ritchie C), is that the programmer's use of parentheses is not honoured on the grounds of efficiency! In floating-point, $(x + y) + z$ is not the same as $x + (y + z)$ and the programmer's parentheses may be inserted specifically to avoid wrong answers. This was corrected when C was Standardised by ANSI: nobody cares how efficiently the wrong answer can be calculated. Many floating-point algorithms fail if optimizing compilers assume 'true real arithmetic'. If optimizing compilers get cleverer then worse problems may occur — see Section 3.2.3 of Goldberg's paper for an example.

Care must be taken when mixing precisions because some of the 'invariance rules' described above do not apply if precisions are mixed. For example, the number 3.0/7.0 is a recurring fraction in base 2 or 10, so it will have different representations in single and double precision. If a compiler evaluates everything in double precision, and x is in single precision, then $x = 3.0/7.0$ may assign the correct value to x, but the test '**if** $(x = 3.0/7.0)$ **then** ...' may not work as expected.

There may be conflicts between IEEE and language Standards. For example, one ANSI language Standard states that 'any arithmetic operation whose result is not mathematically defined is prohibited'. This raises problems because $\pm\infty$ can be used as an ordinary value and may give rise to anomalous results, especially if the operation x/∞ gets rid of the infinite value.

Under default rounding, $x - x$ is defined to evaluate to $+0$ for all finite x. Thus $(+0) - (+0) = +0$. Also $-(+0)$ is defined to be -0, so $-x$ should not be implemented as $0 - x$ as this would be wrong for $x = +0$.

The use of $NaNs$ violates many of the desirable 'invariance rules'. The test '**if** $(x = x)$ **then** ...' yields *true* unless x is a NaN in which case it is always *false* by definition. Also, the test '**if** $(x \leq y)$ **then** ...' is not always the same as '**if** $not(x > y)$ **then** ...' because $NaNs$ are unordered by definition.

Exception handling leads to problems with synchronicity on pipelined machines and with optimizing compilers. Hardware and software solutions to these problems exist but, unfortunately, these are not generally portable to other systems.

3.4 Problem formulation and user interfaces

In the design of a subroutine for general use, say as a part of a numerical software library, there are a number of considerations besides the choice of the underlying algorithm. We will now examine some of these considerations by means of examples.

---

† N.B. Proving the correctness of a floating-point algorithm does not imply that it solves the corresponding mathematical problem correctly. Numerical analysis is still necessary to understand the properties and limitations of the method on which the algorithm is based. For example, it would be very useful to prove the correctness of an adaptive integration procedure (see Section 3.4.1) but, for any such procedure, there must exist functions for which it gives arbitrarily inaccurate answers. There is no contradiction here: the numerical method is a finite approximation to an infinite process, so cannot in principle be made to work in all cases. Proving the algorithm correct amounts to showing that the (intrinsically flawed) numerical method has been implemented correctly. Such a proof would be at least reassuring. Nevertheless adaptive integration methods are useful in practice except only in pathological cases.

3.4.1 Automatic quadrature

Suppose that

$$integrate(f, a, b, eta\_t, result, error)$$

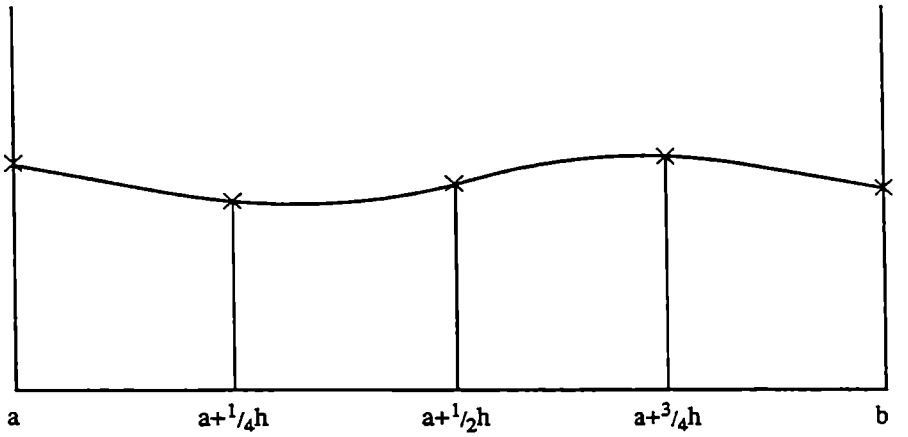
is a subroutine that returns an estimate of $I = \int_a^b f(x)dx$ in *result*, given a target accuracy *eta_t* to be used in a mixed error test. The variable *error* is set non-zero if any kind of error condition is detected (such as detecting that the integral I does not exist).

There is a significant difference between a subroutine that evaluates, say, a certain composite quadrature rule and a subroutine such as *integrate* that performs all necessary calculations and returns an estimate of the integral to a prescribed accuracy. The *black box* approach of the subroutine *integrate* is known as *automatic quadrature*. The algorithm for such a routine not only needs a suitable quadrature rule or rules, but also a strategy for achieving the requested accuracy and a system of bookkeeping to keep track of the progress of the calculation. In particular, if function evaluations are to be minimised, it is necessary to ensure that the function $f(x)$ is not evaluated at the same point more than once, and also that function values are stored if there is a possibility that they may be required more than once. Needless to say, the work done in bookkeeping should be negligible compared with the function evaluations if the algorithm is to be efficient.

Simpson's rule, or the composite form of it, can be used as the basis of more or less sophisticated automatic quadrature schemes. Two such schemes will be described for the integral $\int_a^b f(x)dx$.

Simple automatic Simpson quadrature

A simple strategy is to use the composite Simpson rule with different values of h in such a way that function evaluations can be re-used as much as possible. This can be done by evaluating the rule with an initial value of h then re-evaluating using $h/2$, $h/4$, $h/8$, ... until the process converges to the required accuracy. Consider dividing the integral into first 1 and then 2 panels: this is the simplest case but illustrates the principle involved.



Writing $h = b - a$, the first application of Simpson's rule uses $f(a)$, $f(a + \frac{1}{2}h)$ and $f(b)$. The second application re-uses these 3 values and also needs $f(a + \frac{1}{4}h)$ and $f(a + \frac{3}{4}h)$. The relevant formulae are

$$S_1 = \frac{h}{6}[f(a) + 4f(a + \frac{1}{2}h) + f(b)]$$

$$S_2 = \frac{h}{12}[f(a) + 4f(a + \frac{1}{4}h) + 2f(a + \frac{1}{2}h) + 4f(a + \frac{3}{4}h) + f(b)]$$

with errors given by

$$I - S_1 = -\frac{f^{iv}(\zeta_1)(\frac{h}{2})^5}{90}$$

$$I - S_2 = -\frac{2f^{iv}(\zeta_2)(\frac{h}{4})^5}{90}.$$

If we make the assumption that $f^{iv}(\zeta_1) \simeq f^{iv}(\zeta_2)$ then, by subtracting these error estimates and re-arranging, we get

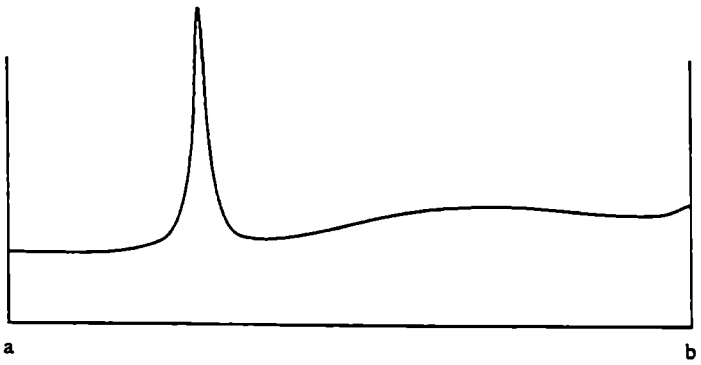
$$\frac{f^{iv}(\zeta_2)(\frac{h}{2})^5}{90} = \frac{2^4}{1-2^4}(S_2 - S_1)$$

from which we get the estimate

$$I - S_2 = \frac{S_2 - S_1}{15}. \tag{3.1}$$

In other words, the difference between the two approximations is about 15 times as big as the absolute error in S_2. Since all the function evaluations used in S_1 are re-used in S_2, the extra work needed to provide this useful error estimate is negligible.

It is clear that a successful and reasonably efficient automatic quadrature routine can be designed using successive halving of the interval h. However, consider how the above scheme would perform when integrating the following function:



In order to achieve greater efficiency, particularly for difficult integrals, it is better to allow the strategy to depend on the function $f(x)$ itself. Such an automatic scheme is said to be *adaptive*.

Adaptive Simpson quadrature

In this scheme the interval (a, b) is first split up into several equal subintervals. Only the two rules S_1 and S_2 are used but, if the estimated contribution to the error in each subinterval is too large then that subinterval is further sub-divided and the process repeated. The net effect is that the final sub-intervals are not all of equal size, and these sizes depend on the behaviour of the integrand $f(x)$ in that region.

The bookkeeping has to take account of all the nested subintervals and to ensure that the whole of (a, b) is covered. There is obviously a lot more work involved in ensuring that function evaluations are re-used, but adaptive schemes are generally more efficient for difficult problems.

Exercise 3a

> Describe how the adaptive Simpson method could be implemented using a recursive function (used internally by *integrate*). How would you ensure that the algorithm terminates? Describe circumstances in which you would expect an adaptive scheme to be (a) more efficient, and (b) less efficient than a non-adaptive scheme.

3.4.2 Ordinary differential equations

Suppose a particular algorithm can be applied to a wide class of problems. If a subroutine is to use this algorithm then it is highly desirable that the user interface should not prevent the solution of problems for which the algorithm is applicable.

This ideal is not always easily achieved, but consider the *first order differential equation*

$$y' = f(x, y); \quad y(a) = b$$

where $y' \equiv dy/dx$ and a and b are given constants. As the numerical solution of the equation begins at the point $x = a$, this is called an *initial value problem.* There are several algorithms in current use for the solution of such equations. Using the notation $x_n = a + nh$, $y_n \simeq y(x_n)$, one of the simplest is the *Runge-Kutta scheme*

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + h, y_n + k_1) \\ y_{n+1} &= y_n + \frac{1}{2}(k_1 + k_2) \end{aligned}$$

which can be used to advance the solution by fixed steps of length h in the x direction‡.

This algorithm, and many other algorithms for this problem, can be generalised to the solution of a set of m simultaneous differential equations of the form

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}); \quad \mathbf{y}(a) = \mathbf{b} \tag{3.2}$$

where a is a value of the scalar x, $\mathbf{b}$ is the initial value of the vector of solutions $\mathbf{y}$, and $\mathbf{f}$ is a vector of functions. This is called a *first-order system* of differential equations. The generalisation of the Runge-Kutta scheme is

$$\begin{aligned} \mathbf{k_1} &= h\mathbf{f}(x_n, \mathbf{y_n}) \\ \mathbf{k_2} &= h\mathbf{f}(x_n + h, \mathbf{y_n} + \mathbf{k_1}) \\ \mathbf{y_{n+1}} &= \mathbf{y_n} + \frac{1}{2}(\mathbf{k_1} + \mathbf{k_2}) \end{aligned}$$

where $\mathbf{k_1}$, $\mathbf{k_2}$ are also now vectors. Because of the similarity of the scalar and vector formulae it is usual for subroutines for initial-value problems to be written for general systems rather than just the scalar case.

In fact, this algorithm is quite generally applicable. Consider, for example, the solution of the second order differential equation

$$y'' + p(x)y'^2 = q(x); \quad y(0) = u, y'(0) = v \tag{3.3}$$

where p, q are given functions and u, v are given constants. By writing $z_1 = y$, $z_2 = y'$ we can write (3.3) as the pair of equations

$$\begin{aligned} z_1' &= z_2 \\ z_2' &= -p(x)z_2^2 + q(x) \end{aligned}$$

with initial conditions $z_1(0) = u$, $z_2(0) = v$. Note that this can be written in the vector form (3.2), and is now a first-order system. This formulation can easily be used for most initial-value problems.

3.4.3 Modular use of software.

Given a reliable source of numerical software for a reasonably wide range of problems, it is often possible to solve more complex problems by combining library subroutines in a suitable way. Such ingenuity does not remove the need for adequate analysis of the algorithms involved but, often, a means to an end can

‡ This algorithm is chosen for illustrative purposes only, so the details do not matter here. The derivation of Runge-Kutta schemes is dealt with in Conte & de Boor.

be justified if the final results can be checked independently. For example, when solving equations, if the computed solutions can be shown to satisfy the equations (to appropriate accuracy) then the computed solutions can be assumed to be adequate.

As an example, it is unlikely that a library subroutine would be found specifically for the problem of finding a vector $\mathbf{x}$ that yields

$$\min_{\mathbf{x}} \int_a^b f(\mathbf{x}, t)dt. \tag{3.4}$$

However a subroutine like *integrate* (as described in Section 3.4.1) would usually be available in a general-purpose library. Suppose also that the subroutine

$$minimise(x, n, g, error)$$

is available as a *black box* for minimising a function $g(\mathbf{x})$, where $\mathbf{x}$ is a vector of length n. The vector $\mathbf{x}$ is held in the array x. As in *integrate*, the variable *error* indicates the success of the algorithm or otherwise. The user initialises the array x with a starting approximation and, if the algorithm is successful, the minimising vector is returned in the same array. The user also must provide code for evaluating the function g. The manner in which this is done depends, to some extent, on the programming language used and could be expected to be part of the specification of *minimise*. This means that the user cannot vary this interface. For the sake of simplicity, let us suppose that g is to be evaluated by means of a subroutine

$$g(x, n, funval, error)$$

where the value of $g(\mathbf{x})$ is returned in the variable $funval$. The values of the function f are also required by *integrate* and could be supplied by a subroutine with the specification

$$f(t, funval, error)$$

where t is the variable over which the integration is to be performed.

The problem (3.4) can be solved by calling *minimise* and providing a subroutine g which itself calls *integrate*; *integrate* calls f. The array x, which is not a part of the specification of f, needs to be made globally available. Note also that, as the value of g for an arbitrary vector $\mathbf{x}$ depends on the success of *integrate*, it is highly desirable that g can return a value of *error* in case *integrate* fails.

Exercise 3b

> Choose any suitable programming language and write brief user documentation on how to implement the method of Section 3.4.3, assuming that appropriate library software is available.

3.5 BLAS: Basic Linear Algebra Subroutines

Finally, we will look briefly at a method currently used to enhance the portability of numerical library software.

Linear algebra is the name usually given to computations involving matrices, including the solution of simultaneous linear equations and eigenvalue problems. Since 1979 attempts have been made to define subroutine interfaces† for basic vector and matrix operations (e.g. to add two vectors, or to multiply a row of a matrix by a vector) in order that more complex algorithms (e.g. Choleski factorization) can be coded in terms of these building blocks. These have been called by the acronym *BLAS*: *B*asic *L*inear *A*lgebra *S*ubroutines. There are currently about 100 specified BLAS.

† Due to C. Lawson, R.J. Hanson, D. Kincaid & F. Krogh, 1979; J.J. Dongarra, J.J. du Croz, S.J. Hammarling & R.J. Hanson, 1988; J.J. Dongarra, J.J. du Croz, I.S. Duff & S.J. Hammarling, 1990; and D.S. Dodson, R.G. Grimes & J.G. Lewis, 1991.

The idea has two significant advantages:

(1) New linear algebra algorithms can be implemented more easily.

(2) On special architectures, such as vector or array processors, algorithms using BLAS can be speeded up by re-coding only the BLAS to take advantage of the architecture.

There are also two (relatively minor) disadvantages worth mentioning:

(1) In problems with a small matrix size, e.g. 2 or 3-dimensional graphics, the use of BLAS may mean that subroutine calls dominate the computation time. Consequently the use of BLAS may be relatively inefficient. (This comment is obsolescent as processors become faster.)

(2) The detailed specification of BLAS is, to some extent, language-dependent and a *de facto* standard specification needs to be established for each appropriate language.

In 1990 the idea of BLAS was extended by providing a portable library of higher level numerical linear algebra routines, called LAPACK, specifically aimed at high-performance computers.